Microsoft Editor User's Guide

Microsoft

827/1

Microsoft® Editor

USER'S GUIDE

FOR MS® OS/2 AND MS-DOS®
OPERATING SYSTEMS

MICROSOFT CORPORATION

Printed and bound in the United States of America.

Document No. LN0801B-500-R00-0889
Part No. 07823
10 9 8 7 6 5 4 3 2 1

Table of Contents Overview

Appendixes

Glossary

Index

Table of Contents

Appendixes

Figures

Tables

CHAPTER 1 Introduction

The Microsoft® Editor is the first full-screen editor to run under both OS/2 systems and DOS (Versions 2.1 and above). You can use the editor to write programs, modify text files, and, under OS/2, run language translators and other utilities in the background.

The Microsoft Editor was specially developed as a programmer's editor. It offers an unsurpassed ability to work efficiently with many different files in different directories, to interact with the environment, and to undo or redo a whole series of commands. As a programmer, you can work faster and more efficiently by using some of these features:

- **Compile and link programs from within the editor**

  Improve your productivity. The Microsoft Editor is more than a text editor; it is a development environment. Develop programs more quickly by compiling from within the editor. If the compilation fails, view the errors, rewrite the program, and recompile—all without leaving the editor.

- **Create new editing functions in C or assembly language**

  Extend the editor's power by writing new functions. If you know how to program in C or assembly language, you can quickly learn how to write new modules for the Microsoft Editor. These modules involve no preprocessing; they become part of the editor itself and therefore run as fast as standard editing commands.

- **Customize the editor to suit your needs**

  Control how the editor behaves. The editor uses a special initialization file, TOOLS.INI, in which you can easily specify your own preferences for function keystrokes, screen colors, tabs, margins, and many other kinds of editor behavior. You can even specify preferences specific to type of file so that as you move between .C and .FOR files, the editor alters its settings.

This manual is a substantial revision of the *Microsoft Editor User's Guide* for Version 1.0 of the editor. This manual has an expanded index, more examples, detailed instructions for configuring the editor, and complete documentation for writing editing functions in C.

A Note about Operating-System Terms

Microsoft documentation uses the term "OS/2" to refer to the OS/2 systems—Microsoft Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" refers to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features unique to that system.

1.1 Editing Capabilities

Not only can the editor be customized, but it also supports many powerful editing features:

- **Use a full range of file-editing commands**

  The editor supports a comprehensive range of file operations. You can load, merge, or save files, with or without exiting. You can execute the DOS or OS/2 shell, send highlighted areas to the printer, or insert program output directly into a file. The editor also supports a wide range of pattern-search and replacement functions.

- **Save time with powerful block operations**

  You can manipulate different kinds of text blocks. For example, you can insert or delete ordinary sequences of characters between two file positions. You can also insert or delete rectangular areas, called "boxes." Box-shaped regions are highly useful for indenting paragraphs or moving columns of text.

- **Save typing effort with macros**

  The Microsoft Editor includes a convenient macro language. A macro is a command that performs a series of predefined actions; for example, a macro can insert a given phrase or word or perform an entire series of editing commands. Define a macro, then invoke it with one keystroke.

- **Edit complex files with windows**

  When editing a large file, you may want to view different parts of the file simultaneously. With the Microsoft Editor, you can split up your screen into as many as eight windows—each displaying a different part of the file (or parts of different files).

- **Handle multiple source files**

  A simple command switches you among the files you are working on—you never have to leave the editor and start it up again to work on a different file. As the editor moves between files, it saves the last cursor position and other relevant information.

1.2 System Requirements

To use the Microsoft Editor, you need OS/2 1.0 or DOS 2.1 or later and at least 128 kilobytes (K) of available memory. A minimum of 150K of available memory is required to use the C extensions described in Chapter 8.

1.3 Using This Manual

The following list gives the chapter or section you should read to learn about a particular topic:

Topic	Chapter or Section
Using the editor right away	Chapter 2, "Edit Now"
Starting the editor using options and file names	Section 2.5, "The Microsoft Editor's Command Line"
Entering arguments to editing functions	Chapter 3, "Command Syntax"
Using the most common editing commands, including file operations, compiling, and windows	Chapter 4, "A Survey of the Editor's Commands"
Using regular expressions with the editor	Chapter 5, "Regular Expressions"
Customizing the editor by changing function-to-key assignments and basic conditions, such as screen colors	Chapter 6, "Function Assignments and Macros," and Chapter 7, "Switches, Assignments, and the TOOLS.INI File"

Writing new editing functions in C or assembly language	Chapter 8, "Programming C Extensions," and Chapter 9, "C-Extension Functions"
Getting quick reference to commands and switches	Appendix A, "Reference Tables," and the index
Using the accompanying utilities: UNDEL and EXP	Appendix B, "Support Programs for the Microsoft Editor"

1.4 Introducing the Microsoft® Editor

You'll find the Microsoft Editor easy to use once you understand the function-argument model it uses for executing commands. Each command consists of a function that may or may not be given input in the form of an "argument."

The Microsoft Editor uses a "reverse Polish" command structure. That is, you specify the argument first, then give the command function. For example, if you want to load a file, you first specify the file name, then invoke the file loading function.

All Microsoft Editor functions work in this fashion. This arrangement is different from that of conventional editors where you would choose the file loading function first, then specify the desired file. This "argument first, function second" sequence gives the Microsoft Editor unique advantages:

- A single function can process different types of arguments, such as highlighted blocks of text, a word or number typed on the dialog line, or the word at the current cursor position. Almost all functions work with all data types, so there aren't too many arbitrary details to memorize.
- It's easy to write macros because almost every action can be expressed as a function name. Macros are likewise easy to understand—you won't have trouble understanding another person's macro, or a macro *you* wrote six months ago.

Function names follow a consistent pattern. For example, functions that involve forward movement usually start with P (for "plus"). Functions that involve backward movement usually start with M (for "minus").

Before using the Microsoft Editor, run the installation program for your Microsoft language product. The installation program sets up the powerful on-line Help, which provides on-line information for editing functions and the standard C library. It is recommended that you select the default keystroke configuration in order to work through the early chapters of this manual.

NOTE *If you choose to customize the Microsoft Editor during installation, its function-key assignments will match those of BRIEF®, Epsilon™, or the Microsoft "Quick" Environment. This book describes the uncustomized version of the Microsoft Editor, so the commands described will not always match those on your version of the editor.*

1.5 Document Conventions

The following document conventions are used throughout this manual and apply in particular to syntax displays for commands and switches:

Example of Convention	Description
AddFile	Boldface type always marks standard features of programming languages (keywords, operators, and functions) and editor switches.
`$INIT: tools.ini`	This font is used to indicate all example programs, user input, and screen output.
placeholders	Words in italics indicate a Microsoft Editor function, a field, or a general kind of information; you must supply the particular value. For example, *numarg* represents a numerical argument that you type in from the keyboard. You could type in a number, such as `15`, but you would not type in the word "*numarg*" itself.
Repeating elements...	Three dots following an item indicate more items having the same form may appear.

`Program`
`.`
`.`
`.`
`Fragment`
	A column of three dots tells you part of a program has been intentionally omitted.

SHIFT	Names of keys on the keyboard appear in small capital letters. Notice that a plus (+) indicates a combination of keys. For example, CTRL+E means to hold down the CTRL key while pressing the E key. The names of the keys in this manual correspond to the key names printed on the IBM Personal Computer keyboard. If you are using a different machine, these keys may have slightly different names.

The cursor-movement keys (sometimes called "arrow" keys) located on the numeric keypad to the right of the main keypad are called the DIRECTION keys. Individual DIRECTION keys are called either by the direction of the arrow on the key top (LEFT, RIGHT, UP, DOWN) or the name on the key top (PGUP, PGDN).

Some of the Microsoft Editor's functions use the +, –, or number keys on the numeric keypad rather than the ones on the top row of the main keyboard. At each instance, the text notes the use of keys from the numeric keypad.

The carriage-return key, sometimes unnamed but marked with a bent arrow, is called the ENTER key.

"Commands"

The first time a new term is defined, it is enclosed in quotation marks.

CHAPTER 2

Edit Now

This chapter shows you how to use the Microsoft Editor right away by focusing on the functions you need to create a simple text file. "Functions" are built-in editing capabilities you invoke to perform actions. Most of the chapter consists of a tutorial that uses a specific example and features the following functions:

Function	Default Keystroke
Cursor movement	DIRECTION keys, HOME, END
Insertmode	INS
Delete	DEL
Arg (introduce argument)	ALT+A
Cancel	ESC
Undo	ALT+BKSP
Paste	SHIFT+INS
Psearch (forward search)	F3
Linsert (insert new line)	CTRL+N
Exit	F8
Help	F1

You can use this tutorial either by starting the editor and typing in each command as shown, or you can simply read along. Because the results are explained at each stage, you can get a good understanding of the editor just by reading.

The chapter ends by presenting the complete command line for the editor with all the possible options you may use, a complete list of items on the status line, and some hints for learning the editor.

2.1 Starting the Editor

Copy the file M.EXE into your current directory or a directory listed in the PATH environment variable. To run the editor in OS/2 protected mode, copy the file MEP.EXE. (You may want to rename the file as M.EXE.) Then start the editor with this command:

```
M NEW.TXT
```

The Microsoft Editor responds by asking if you want to create a new file with this name. Press Y to indicate yes. The editor creates the file and places it in the current directory. You are now ready to enter text.

NOTE *If the editor cannot start correctly, it reports an error message. If you receive an error message on start-up, consult the list of error messages in Appendix C. This list of error messages provides explanations, along with suggestions for solving the problem.*

When you want to exit, press F8. Whenever you want to save your work without exiting, type the following keystrokes:

ALT+A ALT+A F2

2.2 The Microsoft Editor's Screen

When starting the editor with a new file, you see a blank screen, as shown in Figure 2.1 below.

The cursor first appears at the upper-left corner of the screen. Even though the file is empty, you can use the DIRECTION keys—denoted as UP, DOWN, LEFT, and RIGHT—to move the cursor anywhere on the screen. (The DIRECTION keys are the arrow keys on the numeric keypad. Newer keyboards may have an additional set of arrow keys to the left of the numeric keypad.) Try experimenting with cursor movement.

NOTE *The DIRECTION keys on the numeric keypad do not respond unless NUMLOCK is off. Press the NUMLOCK key to toggle the numeric keypad lock on and off.*

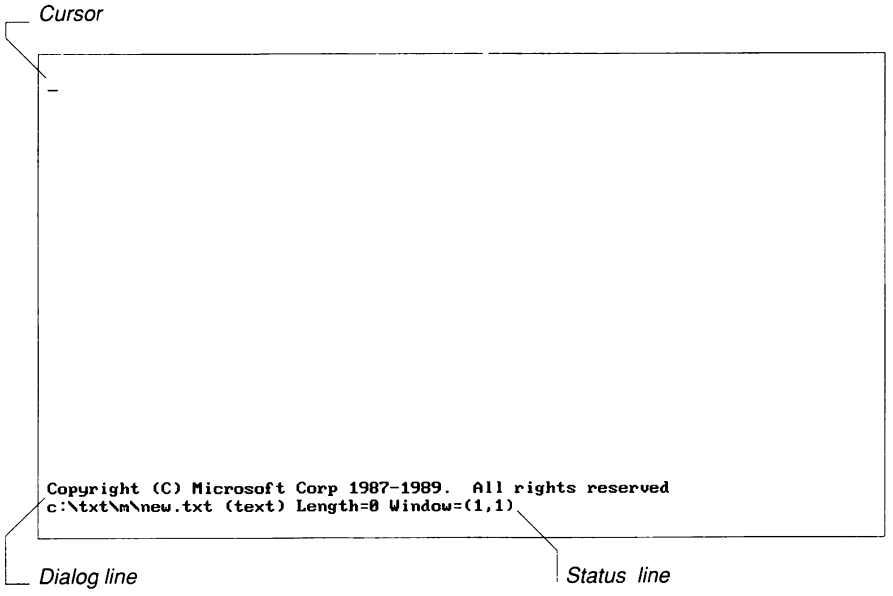


Figure 2.1 Microsoft Editor's Screen

The next-to-bottom line is called the "dialog line," which is reserved for displaying messages from the editor and letting you enter text arguments. The bottom line is called the "status line." It always displays the following fields:

Field	Description
`c:\m\new.txt`	File name, with complete path
`(text)`	Type of file
`Length=1`	Length of file, in number of lines (minimum value is 1)
`Window=(1,1)`	Window or cursor position

The field `Window=(1,1)` indicates that the upper-left corner of the screen corresponds to the first row and column of the file. As you scroll through files larger then one screen, the numbers in this field change. See Section 7.2.1, "Changing Start-Up Conditions," to learn how to alter this field so that it displays cursor position instead of window position.

2.3 Sample Session

Once the Microsoft Editor is started, you can enter text immediately. Simply start typing and press ENTER when you want to begin a new line. By default, the editor starts in "overtype" mode, which means that anything you type replaces the text at the cursor position.The editor also has an "insert" mode for adding new material without replacing the current text. Insert mode is explained in the next section.

To begin, type in the following text. There are some intentional errors you'll correct in a few moments.

```
It's mind over matter.
What is mind?
No mat matter.
Wh is matter?
Mever mind._
```

The third, fourth, and fifth lines have errors near the beginning of each line. To get to the beginning of the fifth line, you can press the LEFT key until you move to the beginning of the line. However, you can get there faster by pressing the HOME key. This key moves the cursor to the first nonblank character in the line.

Now move the cursor to the beginning of the fifth line and correct the error by typing the letter `N`:

```
Never mind.
```

2.3.1 Inserting Text with the Insertmode Function

To insert text in this example, move the cursor to the third position in the fourth line:

```
Wh_is matter?
```

The letters `at` need to be inserted at the end of the first word. Press the INS key to invoke the *Insertmode* function, which toggles between overtype and insert mode. You'll see the word `insert` appear at the end of the status line. Type the letters `at` to produce the following line:

```
What_is matter?
```

To return to overtype mode, press INS again.

2.3.2 Removing Text with the Delete Function

So far, you've used editing functions to replace old text and insert new text. The third line requires text deletion, so move the cursor to the beginning of the second word in the third line:

```
No mat matter.
```

Invoke the *Delete* function by pressing the DEL key. Each time you press the DEL key, the character at the cursor position is erased.

It is inconvenient to delete a large amount of text one character at a time. The *Delete* function can also remove blocks of text that are specified as arguments. The next section explains deleting a large block of text.

2.3.3 Using the Arg Function to Specify Text

The *Arg* function is assigned to ALT+A; hold down the ALT key and press A to invoke it. The *Arg* function does nothing by itself. Rather, it tells the editor that you are about to introduce an argument that another function will operate on. (An "argument" is input, such as text or highlighted characters.)

In this example you'll use the *Arg* function to highlight the group of characters you wish to delete. After pressing ALT+A, move the cursor to the beginning of the third word, as shown in Figure 2.2 below. Now press DEL, and the highlighted characters are removed.

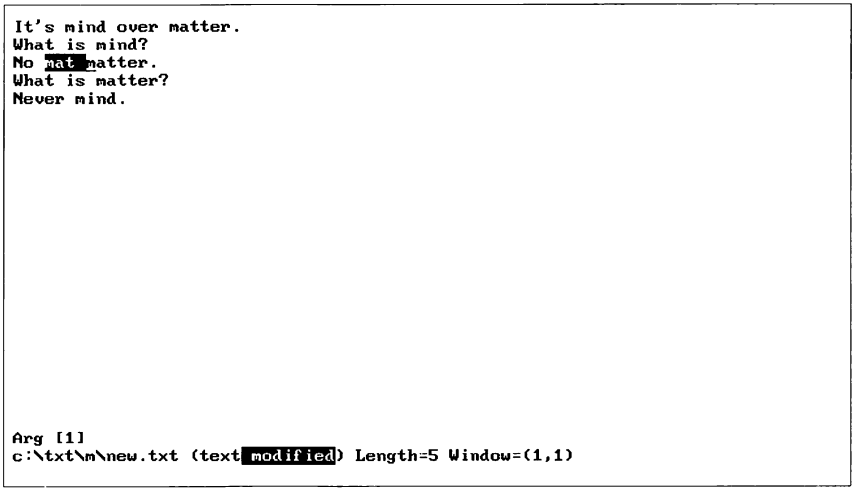


Figure 2.2 The *Arg* Function

2.3.4 Canceling and Undoing Commands

If you pressed ALT+A at the wrong time but did not complete the command you were typing, you can cancel the argument by pressing the ESC key. This keystroke invokes the *Cancel* function. The *Cancel* function lets you start a command sequence over again.

If you complete an incorrect command, reverse it by pressing ALT+BKSP (hold down the ALT key and then press the backspace key). This keystroke invokes the *Undo* function. If you invoke *Undo* again, it reverses the next-to-last editing command. Invoke *Undo* a third time, and it reverses the second-to-last editing command, and so on. The number of commands that the editor remembers is controlled by the **undocount** switch. The default number of commands remembered is 10. See Chapter 7, "Switches, Assignments, and the TOOLS.INI File," for information on how to set switches.

The *Meta* function (F9) is a prefix, similar to *Arg*, which reverses or alters the effect of a function. You can cancel the effect of the *Undo* function with the command sequence *Meta Undo*:

F9 ALT+BKSP

This variation on the *Undo* function is often called "Redo." If you undo a series of commands, you can recall each of these commands by using *Meta Undo* repeatedly. *Undo* walks backward through the history of the file (it restores an earlier state); *Meta Undo* walks forward.

2.3.5 Using Delete to Move Text

The *Delete* function can be used to move text as well as delete it. The last text deleted is placed on the "Clipboard." The Clipboard holds text selected by either the *Copy* or *Delete* function. Pressing SHIFT+INS invokes the *Paste* function, which inserts the contents of the Clipboard into the file at the present cursor position.

In this section, *Delete* will be used to move two complete lines of text. Consider the current text:

```
It's mind over matter.
What is mind?
No matter.
What is matter?
Never mind.
```

Move the cursor to the beginning of the fourth line. Select the bottom two lines by pressing ALT+A and then pressing the DOWN key twice. You should see the bottom two lines highlighted, as shown in Figure 2.3 below.

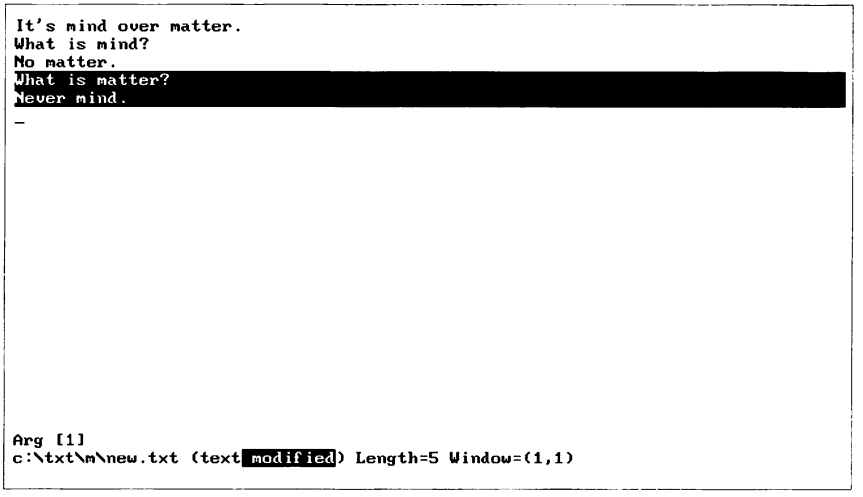


Figure 2.3 The *Ldelete* Function

Now invoke the *Delete* function by pressing DEL. The two lines disappear. The *Delete* function deletes the characters you highlight.

Having deleted a block of characters, you are ready to use the *Paste* function (SHIFT+INS) to insert the deleted text at a new location. Move the cursor to the beginning of the top line and press SHIFT+INS. You should see the following text:

```
What is matter?
Never mind.
It's mind over matter.
What is mind?
No matter.
```

You can change the shape of the highlighted region by using the *Boxstream* function (CTRL+B), which changes the region into a rectangular area. See Chapter 3 for more information.

2.3.6 Finding Strings with the Psearch Function

The *Psearch* function takes different kinds of arguments but performs the same general operation with each—searching for a string of text. The term *Psearch* stands for "plus search," and means the same thing as "forward search." This function, which is assigned to the F3 key, takes both text arguments and cursor-movement arguments. You can ask the editor to locate the next occurrence of the

word `mind` by typing the word in as a text argument. Move the cursor to the beginning of the file, then try the following sequence of keystrokes:

1. Press ALT+A
2. Type the following text: `mind`
3. Press F3

As you type the word, it appears on the dialog line—the line just above the status line at the bottom of the screen. As soon as you press ALT+A and type the first character, the prompt `Arg:` appears on the dialog line. You can retype characters on the dialog line by pressing BKSP (the backspace key).

You can achieve the same result by moving the cursor to the beginning of the word `mind` on the screen, then highlighting the word with the following sequence of keystrokes:

ALT+A RIGHT RIGHT RIGHT RIGHT F3

An even easier way of selecting the word is to give the keystroke sequence ALT+A F3, which selects the word at the current cursor location. This word (all characters up to the first blank or new-line character) becomes the search string.

Often when you use the *Psearch* function, you want to look repeatedly for some text string. To search for the text string most recently specified, press F3 by itself.

2.3.7 Inserting Spaces and Lines

You'll sometimes need to insert blank lines. There are two ways to do this:

1. Press ENTER. When insert mode is on, pressing ENTER inserts a new-line character. (Recall that you turn insert mode on and off by pressing INS.) Any text to the right of the cursor moves down into the newly created line.
2. Invoke *Linsert* (line insert) by pressing CTRL+N. Regardless of cursor position, the *Linsert* function inserts a blank line directly above the current line.

Linsert also accepts an argument. Invoke *Arg* (ALT+A), move the cursor, and then invoke *Linsert* (CTRL+N). *Linsert* inserts as many blank spaces in front of the highlighted area as the number of characters you highlighted.

2.3.8 Exiting the Editor

Press F8 to leave the editor. The F8 key sequence invokes the *Exit* function, which automatically saves any changes you have made to the file and exits. The sequence F9 F8 exits without saving, and the sequence ALT+A ALT+A F2 saves your most recent changes to the file without exiting.

See Section 4.1, "Basic File Operations," for more information on loading and saving files.

2.4 Getting Help

One of the outstanding features of the editor is on-line Help. On-line Help displays the information you need directly on the screen without your having to exit the editor. You can quickly get information about any editing function.

To use the basic on-line Help, you must first run the installation procedure for the editor and follow all directions. The setup program for this Microsoft language product should install the editor for you. Installing the editor is important because it copies the Help files to your directories and properly configures the editor's initialization file, TOOLS.INI.

2.4.1 Starting On-Line Help

Press SHIFT+F1 to bring up the first Help screen. On-line Help is largely self-explanatory. The first screen gives a list of the basic Help commands and provides access to other parts of the Help system.

Press F1 for context-sensitive Help. "Context-sensitive Help" displays a Help screen for the word at the current cursor position. The entire word is used—even if the cursor is in the middle of the word. If the cursor is on a space, the Help system uses the word immediately preceding the space. Thus you can type a word, press F1, and get Help for that word.

The editor searches through its list of Help files for a screen corresponding to the word at the cursor position. If no screen exists for the topic, the editor displays the first Help screen.

2.4.2 Moving through On-Line Help

The first Help screen explains how to move between Help screens on different topics. To move around within a Help topic, use the same keys (DIRECTION keys, HOME, END, F3) that you use to move through a file. You can use PGUP and PGDN to move up or down a page at a time.

2.4.3 Leaving On-Line Help

By default, the editor splits the screen into two windows when you bring up a Help screen. You can move between windows by pressing F6. Moving from the Help window to the editing window lets you continue editing while still viewing Help information. To close the Help window, press ESC.

Section 7.2.5, "Changing the Look and Feel of Help," explains how to use Help without splitting the screen. If you use this technique, you leave Help by pressing F2 or ESC.

2.5 The Microsoft Editor's Command Line

Use the following command line to start up the editor (the options are case sensitive):

M [[/D]] [[/e *command*]] [[/t]] [[*files*]]

Begin the command line with the base of the editor's actual file name. For example, if you are using the protected-mode version, the editor's file name is MEP.EXE. (However, you can rename this file to M.EXE.)

The /D option prevents the editor from examining TOOLS.INI for initialization settings (see Chapter 7, "Switches, Assignments, and the TOOLS.INI File," for more information).

The /e option enables you to specify a command upon start-up. The *command* argument is a string that follows the same syntax rules as those given for macros in Chapter 6, "Function Assignments and Macros." If *command* contains a space, the entire string should be enclosed in quotation marks. To represent embedded quotation marks, precede each quotation mark by a backslash (\"). To represent a literal backslash, use two backslashes (\\). For example, the system-level command

```
M /e "arg \"search \\\" string\" psearch" FILE.TXT
```

causes the editor to start up and execute the following command:

```
arg "search \" string" psearch
```

The /t option specifies that the names of any files following this option are not retained in the editor's information file when the session terminates. When you use the *Information* command at the next editing session to list the most recently edited files, these names will not appear.

If a single *file* is specified, the editor attempts to load the file. If the file does not yet exist, the editor asks you if you want to create the file. If you type Y (yes),

the editor creates the file. If you type another character, the editor does not create the file, but loads the most recently edited file or (if no files have been previously edited) exits.

If multiple *files* are specified, the first file is loaded; then, when you invoke the *Exit* function, the editor saves the current file and loads the next file in the list. If no *files* are specified, the editor attempts to load the file you were editing when you last exited the editor.

You can specify multiple *files* by either typing in different file names explicitly or by using the DOS wildcard characters, ? and *. For example, the following command line causes the editor to load every file in the current directory with a .TXT extension:

```
M *.txt
```

On start-up, the status line displays at least four fields. The status line can display up to thirteen fields. The first four fields listed below are always displayed:

1. Name of the file being edited.
2. Type of file (based on extension).
3. The length of the file in lines.
4. Cursor position or window position of upper-left corner.
5. The word `modified` if the file has been changed.
6. The letters `NL` if no carriage returns were found when the file was loaded (that is, if the file did not contain carriage returns to denote the end of each line, but used only line feeds).
7. The word `insert` if you are in insert mode.
8. The word `meta` if you have invoked the *Meta* function.
9. The word `No-Edit` if the file cannot be changed at any time during the editing session. Some of the internal files created by the editor, such as <information> and <assign>, fall into this category.
10. The word `RO-File` if the file has the read-only attribute. (This attribute is set outside of the editor.) The file can be modified, but after being modified, it can be saved only under a different file name.
11. The word `cancel` if you recently invoked the *Cancel* function.
12. The letters `BC` if a background compilation is in progress under OS/2 protected mode (or `XX` if the background compilation failed to begin).
13. The letters `REC` while a macro is being recorded.

2.6 Hints for Using the Editor

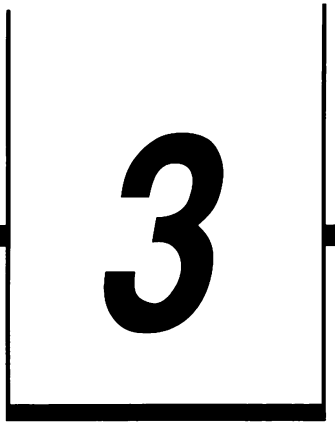
Here are two hints to keep in mind as you learn to use the editor:

1. The Microsoft Editor names functions in a consistent way. Section 3.2, "Naming Conventions for Functions," explains the conventions. Table A.1 lists editing functions by category.

2. If you forget which keystroke is associated with a function, use on-line Help for quick reference. If Help is installed, you can get function assignments by pressing SHIFT+F1 and then choosing the Current Assignments screen. If Help is not installed, pressing F1 takes you directly to this screen.

CHAPTER 3

Command Syntax



If you've worked through Chapter 2, you have seen the flexibility of Microsoft Editor commands. Many of the functions accept a variety of arguments—text arguments, cursor-movement arguments—or no argument at all. This chapter describes each argument type in detail. The chapter also presents the syntax and naming conventions used throughout the manual.

Topics are covered in the following order:

- Entering a command
- Function naming conventions
- Argument types
- Text arguments (*numarg, markarg, textarg)*
- Highlighting a text argument
- Cursor-movement arguments (*streamarg, linearg, boxarg)*

3.1 Entering a Command

Commands take two basic forms. You can invoke a function by itself, or you can introduce an argument and then invoke a function.

Use the *Arg* prefix to introduce an argument. By default, *Arg* is assigned to ALT+A (hold down the ALT key and press A). You can then type characters or move the cursor to highlight part of the screen.

Finally, invoke the function you want by pressing the corresponding keystroke. If you forget the keystroke, you can look it up in on-line Help.

The two basic forms of a command are summarized below:

Function
Arg argument Function

Some functions also let you form a command in one or more of the following three ways:

1. Some functions accept *Arg* with no argument: *Arg Function.*
2. Some functions accept *Arg Arg* (press ALT+A twice). The sequence *Arg Arg* can introduce a function just as *Arg* does.
3. Some functions work differently when given the *Meta* prefix. You can toggle *Meta* on and off by pressing F9. When *Meta* is on, the word `meta` appears on the status line.

See Chapter 4, "A Survey of the Editor's Commands," and Table A.3 for information on what syntax is accepted by each function.

Once you begin entering a command, you can cancel *Arg* and any argument by invoking the *Cancel* function (ESC). The Cancel function is also useful for clearing the dialog line.

NOTE *Throughout this manual, function names are given in italics and are initial capped (for example: Paste). Argument types are given in italics and are lowercase (for example: textarg).*

3.2 Naming Conventions for Functions

The Microsoft Editor follows a consistent pattern of function names. Function names often begin with the letters P, M, S, L, or Cur, which can have the following meanings:

Initial Letter(s)	Usage
P	Plus. Indicates forward movement of some kind. For example, *Psearch* is the forward-search command, and *Pword* moves the cursor forward one word.
M	Minus. Indicates backward movement of some kind. For example, *Msearch* is the backward-search command, and *Mword* moves the cursor backward one word.

S	Stream. Indicates a stream-oriented, block-of-text function. This category includes *Sinsert*, which inserts spaces in the stream of text between two cursor positions, and *Sdelete*, which deletes the stream. Without an argument, these functions insert or delete a single space.
L	Line. Indicates a line-and-box-oriented function. This category includes *Linsert*, which inserts spaces in the exact area highlighted on screen, and *Ldelete*, which deletes it. Without an argument, these functions insert or delete a line.
Cur	Current. Indicates one of the special insertion functions, such as *Curdate* (insert current date) and *Curfile* (insert current file name).

3.3 Argument Types

There are two basic ways to enter arguments: you can enter text directly as part of the command (text argument), or you can use cursor movement to highlight characters on the screen (cursor-movement argument). Each of these two methods has several variations, as shown in the following list:

1. Text argument. After you invoke *Arg* (ALT+A), continue to type characters. These characters appear on the dialog line (the line next to the bottom of the screen). You can give three different kinds of text arguments:
   a. A *numarg*, which consists of a string of digits.
   b. A *markarg*, which is a string containing the name of a previously defined file marker.
   c. A *textarg*, which is any text argument not recognized as a *numarg* or *markarg*.
2. Cursor-movement argument. After you invoke *Arg* (ALT+A), the current cursor position is highlighted. Highlight more characters by moving the cursor to a new position. You can give three different kinds of cursor-movement arguments:
   a. A *linearg*, in which the old and new cursor positions are in different lines but the same column.
   b. A *boxarg*, in which the old and new cursor positions are in different columns.
   c. A *streamarg*, which can consist of any cursor movement.

3.4 Text Arguments (numarg, markarg, textarg)

After you invoke *Arg* (ALT+A), you can enter a text argument by typing any printable characters, including blank spaces. The first time you invoke *Arg*, the following characters appear on the dialog line (the line next to the bottom of the screen):

```
Arg [1]
```

If you press ALT+A again, the number in the brackets changes. Any characters you type appear on the dialog line after `Arg [1]`.

When entering a text argument, you can edit, move through, or modify the text argument in the following ways:

1. Erase a character by pressing BKSP.
2. Erase the character at the current cursor position with the *Sdelete* function (DEL).
3. Move back and forth nondestructively with LEFT and RIGHT. If you use RIGHT to move past the end of current input, the editor inserts the character from the corresponding position in the previous text argument.
4. Insert a space at the cursor position with the *Sinsert* function (CTRL+J).
5. Move to the beginning of the text with *Begline* (HOME) and to the end of the text with *Endline* (END).
6. Clear characters from the current cursor position to the end of the line with the *Arg* function (ALT+A).

To repeat the most recently entered text argument, invoke the *Lasttext* function, which by default is assigned to CTRL+O. When you use *Lasttext*, the *Arg* prefix is unnecessary.

3.4.1 The numarg Type

A *numarg* is a string of digits that you enter as a text argument. Each of the three following examples is a *numarg*:

```
3
11
45
```

The number must be a valid decimal integer. A *numarg* is evaluated as a number and not as literal text. Typically, it is used to indicate a range of lines starting with the cursor position. For example, the following command sequence deletes 10 lines starting with the cursor position:

1. Invoke *Arg* (press ALT+A)
2. Type the following text: `10`
3. Invoke *Ldelete* (press CTRL+Y)

Some functions accept text arguments but do not recognize a *numarg*. These functions treat a *numarg* as an ordinary *textarg*.

3.4.2 The markarg Type

A *markarg* is a file-marker name you have previously defined with the *Mark* function (CTRL+M). See Section 4.4, "Using File Markers," for information about *Mark*.

Once defined, you can enter the marker name as a *markarg*. The name is not treated as literal text, but is interpreted as an actual file position. For example, the following command sequence copies all text between the cursor position and the file position previously marked as `P1`:

1. Invoke *Arg* (press ALT+A)
2. Enter the following text: `P1`
3. Invoke *Copy* (press CTRL+INS)

Many functions accept text arguments but do not recognize a *markarg*. In these cases, the *markarg* is treated as an ordinary *textarg*.

3.4.3 The textarg Type

A *textarg* is similar to a *numarg* or *markarg*. The only difference is that the *textarg* has no special meaning; it is interpreted by the function as literal text. For example, the following sequence finds the next occurrence of the string `Happy New Year`:

1. Invoke *Arg* (press ALT+A)
2. Type the following: `Happy New Year`
3. Invoke *Psearch* (press F3)

A *textarg* can either be typed in directly or highlighted on the screen. The next section describes how to highlight a text argument.

3.5 Highlighting a Text Argument

If the text argument already appears in the editing screen, you can save typing by highlighting the text. You can highlight all or part of one line; a text argument cannot consist of multiple lines.

To highlight a text argument:

1. Move the cursor to the first character in the text argument.
2. Invoke *Arg* (press ALT+A).

   The current cursor position defines the "initial cursor position." As you move the cursor, characters between the initial cursor position and the new cursor position are highlighted.
3. Move the cursor to the right until all the desired text is highlighted.
4. Invoke the desired function. The editor passes the highlighted characters to this function, just as if you had directly typed in the highlighted characters.

For example, the highlighted area in Figure 3.1 defines the text argument `pascal MoveCur`.

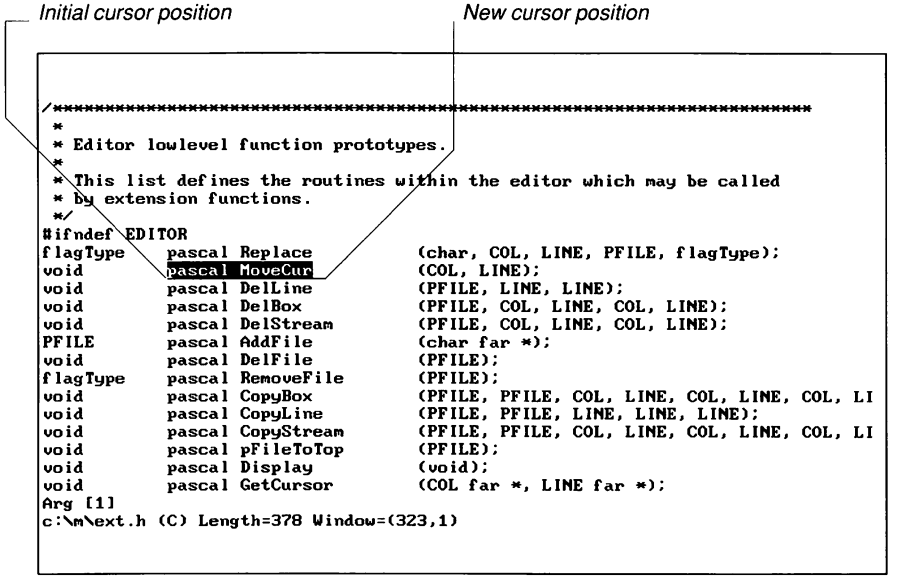


Figure 3.1 Highlighting a *textarg*

Even if the highlighted text represents a legitimate *numarg* or *markarg*, it will be interpreted by the function as a straight *textarg*. That is, a *numarg* or *markarg* must be typed in on the status line; it cannot be selected from the body of text.

Highlighting a *textarg* is a special case of a cursor-movement argument. The next section discusses the other kinds of cursor-movement arguments.

3.6 Cursor-Movement Arguments (linearg, boxarg, streamarg)

You enter a cursor-movement argument by invoking *Arg* (ALT+A) and moving the cursor. When you invoke *Arg*, the current cursor position is marked with a reverse-video highlight. This position is called the "initial cursor position." As you move the cursor, characters between the initial cursor position and the new cursor position are highlighted.

When the initial and new cursor positions are on different lines, you can highlight regions by one of two different modes:

- In "box mode," the editor highlights either complete lines (if the new and initial cursor position are in the same column) or rectangular areas. In box mode, you can select complete lines, or highlight, delete, and insert columns without affecting the surrounding text. In box mode, a cursor-movement argument is either a *linearg* or *boxarg*, as explained in Sections 3.6.1 and 3.6.2.
- In "stream mode," the editor highlights text the way most text editors do. The highlighted region includes all text between the two positions, according to their sequence in the file. This region is usually not rectangular. In stream mode, a cursor-movement argument is always a *streamarg*, as explained in Section 3.6.3, "The streamarg Type."

By default, the editor uses stream mode. You can toggle back and forth between the two modes by invoking the *Boxstream* function (CTRL+B). You can even invoke *Boxstream* while in the middle of creating a cursor-movement argument—doing so changes the highlighting instantly.

To repeat the most recently entered cursor-movement argument, invoke the *Lastselect* function, which by default is assigned to CTRL+U. When you use *Lastselect*, the *Arg* prefix is unnecessary.

If you create a cursor-movement argument and then type a character, the editor removes the highlighted area and replaces it with the character typed. A similar result occurs when you create a cursor-movement argument and then invoke the *Paste* function—the highlighted area is replaced by the contents of the Clipboard.

3.6.1 The linearg Type

A *linearg* is defined when the new cursor position is in the same column but on a different line from the initial cursor position. The editor must be in box mode. The editor responds by highlighting all lines between the two cursor positions, including the lines that the cursor positions are on. For example, the display in Figure 3.2 is produced by invoking *Arg* (ALT+A) and then pressing DOWN three times.

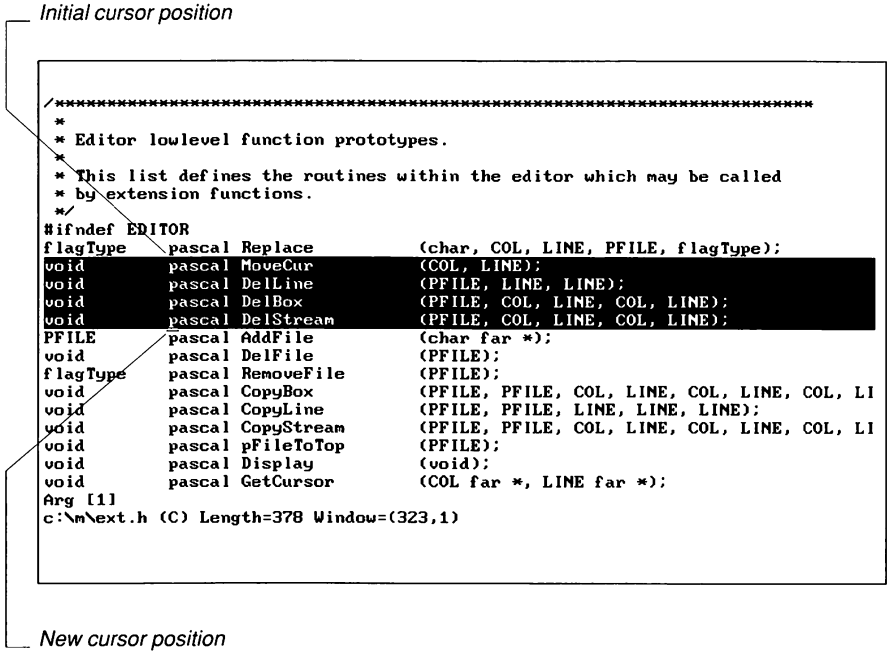


Figure 3.2 Sample *linearg*

3.6.2 The boxarg Type

A *boxarg* is a rectangular area on the screen. The two corners of the area are determined by the initial and new cursor positions. A *boxarg* is defined when the two positions are in different columns (and possibly different lines). The editor must be in box mode.

After invoking *Arg* (ALT+A), you can move the cursor left or right. The left edge of the box includes the leftmost of the two cursor positions. The right edge of the box includes the column just to the left of other cursor positions. The box contains parts of all lines, inclusive, between the two positions.

For example, the display shown in Figure 3.3 is produced by invoking *Arg* and then moving the cursor 3 lines down and 16 columns over.

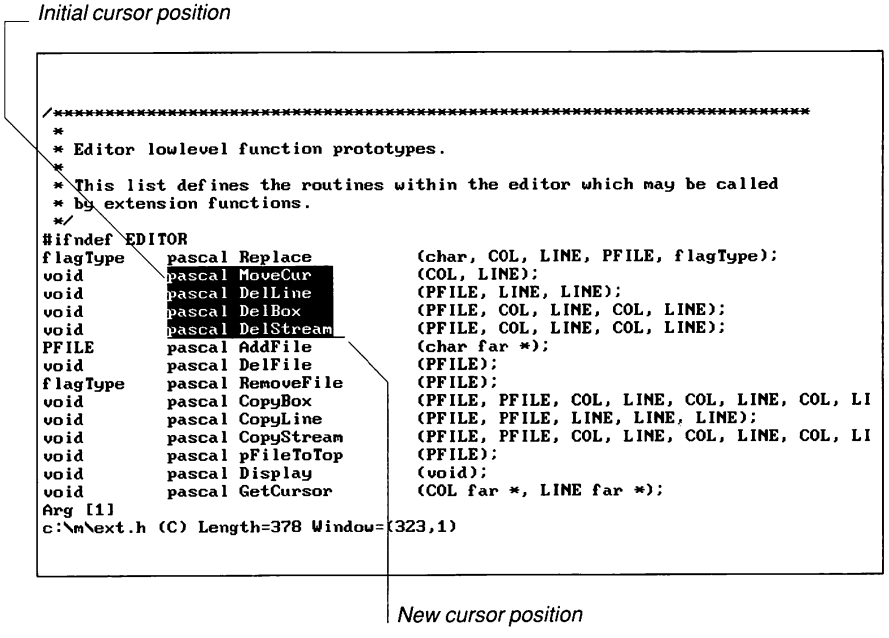


Figure 3.3 Sample *boxarg*

3.6.3 The streamarg Type

A *streamarg* consists of text between the initial cursor and the new cursor positions. When the editor is in stream mode, every cursor-movement argument is a *streamarg*. You toggle between box and stream mode by invoking *Boxstream* (CTRL+B).

After pressing ALT+A, you can move the cursor in any direction to create a *streamarg*. The *streamarg* is shown as highlighted characters.

If you move the cursor forward (that is, to the right or down), the *streamarg* includes the character at the initial cursor position. If you move the cursor backward (that is, to the left or up), the *streamarg* does *not* include the character at the initial cursor position; the *streamarg* starts at the preceding character.

When a *streamarg* spans multiple lines, it includes some characters that a *boxarg* does not. Specifically, a multiline *streamarg* includes the following:

1. All characters from the first cursor position to the end of the line
2. All characters on the lines between the two cursor positions
3. All characters on the line of the second cursor position, up to but not including the cursor position itself

For example, the display shown in Figure 3.4 is produced with the same cursor movement used in Figure 3.3, but with stream mode on.

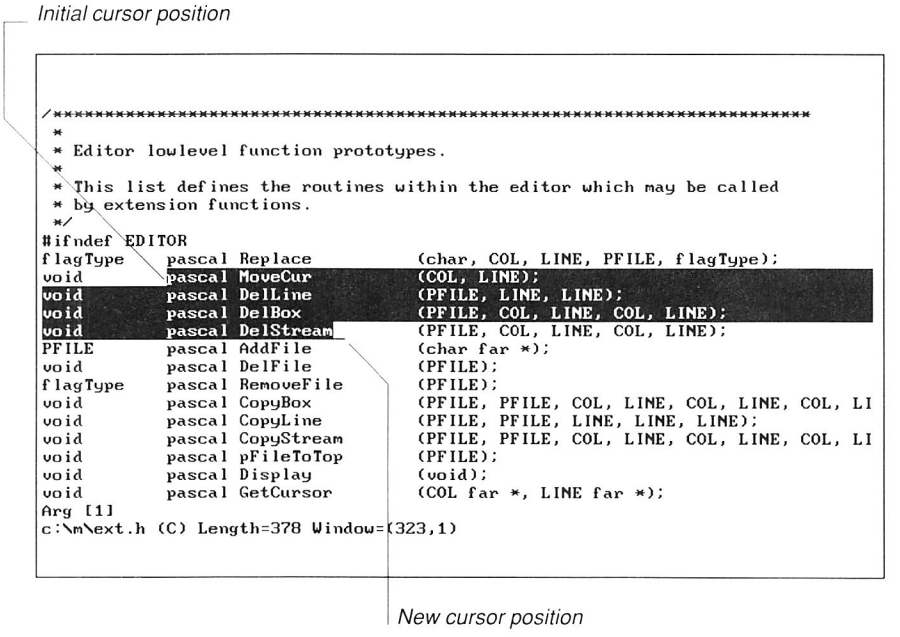


Figure 3.4 Sample *streamarg*

CHAPTER 4

4

A Survey of the Editor's Commands

The Microsoft Editor has all the standard features of a programmer's editor. It lets you move quickly through a file, move blocks of text, search for strings, and handle multiple files. In addition, the Microsoft Editor allows you to use different windows for viewing more than one file or more than one part of the same file. The Microsoft Editor can also invoke compilers and assemblers, then display each compilation error.

This chapter expands on the editing topics introduced in Chapter 2. For a complete list of the command syntax for every function, see Appendix A, "Reference Tables." This chapter covers topics in the following order:

- Basic file operations
- Moving through a file
- Inserting, copying, and deleting text
- Using file markers
- Searching and replacing
- Compiling
- Using editing windows
- Working with multiple files
- Printing all or part of a file

4.1 Basic File Operations

This section discusses how to work with files in general and how to use internal files called "pseudo files."

4.1.1 File Commands

File operations are basic to all work with the editor. You use file operations to save your work, load in a new text file, or completely exit from the editor.

Chapter 2 described how to save the current file and exit from the editor. However, the Microsoft Editor supports a number of other file operations. The list below shows how to use some of the most common file operations:

Command (and Default Keystrokes)	Description
Exit (F8)	Exits editor after saving current file
Meta Exit (F9 F8)	Exits editor without saving
Arg Arg Setfile (ALT+A ALT+A F2)	Saves current file without exiting
Arg Arg textarg Setfile (ALT+A ALT+A *textarg* F2)	Saves current file under the file name (*textarg*) without exiting
Arg textarg Setfile (ALT+A *textarg* F2)	Loads another file (entered as *textarg*) into the editing window
Arg Arg textarg Paste (ALT+A ALT+A *textarg* SHIFT+INS)	Merges (copies) another file (entered as *textarg*) into current file—the new file is inserted at the cursor position
Refresh (SHIFT+F7)	Discards most recent changes to current file and rereads file from disk

Example

As explained in Chapter 3, a *textarg* is simply an argument you type in directly. For example, to load the file SAMPLE.TXT, you would follow these steps:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following file name: `SAMPLE.TXT`
3. Invoke the *Setfile* function (press F2)

Each of the operations listed above is frequently useful, yet some should be used with caution. For example, you should only exit without saving when you have

accidentally altered a file you do not want to change, or when you have made many mistakes during the editing session. When you exit without saving, all the work you did since the last save operation is discarded.

Conversely, you may wish to save without leaving the editor. This operation is usually safe, unless the current file is one that should not be changed. Saving a file writes all the changes you made out to the disk. If your system is vulnerable to power failures or other kinds of system failure, it is a good idea to save your work often.

Merging and loading a file are not the same. Merging a file copies the contents of another file and inserts it into the current file. Loading a file first saves the current file (assuming the **autosave** switch is on), then restarts the editing session with a new file.

The editor supports other variations of these operations described above. See Table A.3, under the *Exit* and *Setfile* functions, for more information.

4.1.2 Special Syntax for Setfile

A text argument passed to *Setfile* can take a number of different forms: a file name, a file name with a DOS wildcard character (* or ?), the name of a directory, or the name of a disk drive. If the text argument is a directory name, the editor changes the current directory. If the argument is a drive name, the editor changes the current drive.

File names can be complete path names and can include environment variables defined with the system-level SET command. The *Setfile* function interprets an environment variables as a list of directories to search for a file. You enter an environment variable using the syntax

$*environ*:*filename*

in which *environ* is the name of an environment variable. For example, the following actions cause the editor to search the INIT environment variable to find the TOOLS.INI file and load it:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following file name: `$INIT:tools.ini`
3. Invoke the *Setfile* function (press F2)

You can also use the environment-variable syntax with the *Paste* function when you use this function to merge a file.

Finally, you can switch to a recently edited file by using a short name. A "short name" is a file name with no path or extension. The editor searches its list of recently edited files to find a name that matches the short name.

4.1.3 Pseudo Files

A "pseudo file" is an internal editing file. It exists only in the computer's memory and does not correspond to any disk file. The editor treats a pseudo file just like any other file except for two important differences:

1. The name of a pseudo file always appears in angle brackets (< >).
2. A pseudo file may not be saved to disk under its own name.

Why use pseudo files? Pseudo files are useful for temporary storage. Pseudo files are updated almost instantaneously because writing to RAM is much faster than writing to disk. You can also insert the contents of a pseudo file into the current file with the *Paste* command.

The editor creates several pseudo files of its own and gives them special meaning. You can open these files just like any other with the *Setfile* function.

Pseudo File	Description
<clipboard>	Stores text selected by a copy or deletion. This file can be modified; the modified file is then inserted when you invoke *Paste*. However, if you copy or delete text while editing this file, the selected text is not saved to the Clipboard but simply discarded.
<assign>	Shows an updated list of function and switch assignments. You can create new function and switch assignments by directly modifying this file; however, the assignments you make must follow the syntax described in Chapter 6, "Function Assignments and Macros."
<information-file>	Shows a list of files that have been previously edited. This file cannot be modified. Each file currently open for editing is listed along with its length in lines.
<compile>	Shows error messages from the last compilation executed from within the editor. If the editor is running under protected mode, the error messages are dynamically updated while the compilation is running.

<file-list>	Shows a list of each file specified on the editor's command line that has not yet been opened. This file cannot be modified.
<record>	Records each editing command when you record a macro. The file is dynamic; it changes while you view it. You cannot directly modify this file. See Chapter 6, "Function Assignments and Macros."

4.2 Moving through a File

Chapter 2, "Edit Now," described how to the use DIRECTION keys to move through a file one space at a time. The DIRECTION keys correspond to the functions *Up*, *Down*, *Right*, and *Left*, to which you can assign different keys if you wish. Chapter 2 also presented the *Begline* function (HOME), which moves the cursor to the first printable character in the current line. Similar to the *Begline* function is the *Endline* function (END), which moves the cursor just to the right of the last printable character in the current line.

Each of the four direction functions in the following list has a variation that uses the *Meta* function as a prefix. Each function, when used in a command with the *Meta* prefix, moves the cursor as far as possible within the displayed screen (or window) without changing column position or causing the screen to scroll.

Command (and Default Keystrokes)	Description
Meta Up (F9 UP)	Moves the cursor to the top of the screen
Meta Down (F9 DOWN)	Moves the cursor to the bottom of the screen
Meta Left (F9 LEFT)	Moves the cursor to the left-most position on the current line
Meta Right (F9 RIGHT)	Moves the cursor to the right-most position on the current line
Meta Begline (F9 HOME)	Moves the cursor to column 1

4.2.1 Scrolling at the Screen's Edge

You can use the four direction functions (*Up*, *Down*, *Right*, *Left*) to cause scrolling. The screen (or current window) can scroll in all four directions. Although the editor does not wrap lines that are wider than the screen, you can have lines of text that are up to 250 characters wide. Use the DIRECTION keys to scroll right and left when your text lines are wider than the screen or current window.

Unlike some editors, the Microsoft Editor does not automatically scroll by only one column or one line. Instead, the internal switches **hscroll** (horizontal-scroll) and **vscroll** (vertical-scroll) control how fast the editor scrolls. For example, if **vscroll** is set to 7, the editor advances the screen position seven lines when you attempt to move the cursor off the bottom of the screen. See Chapter 7 for more information on these switches.

4.2.2 Scrolling a Page at a Time

The editor provides the *Ppage* (PGDN) and *Mpage* (PGUP) functions to move through a file more quickly than you can by using the DIRECTION keys to move one line or one column at a time.

The term "page" is defined as the amount of text that can be displayed in the current window or screen. To advance one page forward through a file, invoke the function *Ppage* (PGDN), which stands for "plus page."

The function *Mpage* (PGUP), which stands for "minus page," is the inverse of *Ppage*, and it moves back through the file one page at a time.

NOTE *In Version 1.0 of the editor, the Ppage and Mpage functions took arguments. You can use them to help build a cursor-movement argument; they do not cancel the current argument. The editor also provides two new functions—Begfile and Endfile—that move to the beginning and end of the file. The next section describes these functions.*

4.2.3 Moving to the Top or Bottom of the File

The *Begfile* and *Endfile* functions provide the fastest cursor movement.

To move the cursor to the beginning of the file, invoke the *Begfile* function by pressing CTRL+UP (the up-arrow key on the numeric keypad).

To move the cursor to the end of the file, invoke the *Endfile* function by pressing CTRL+DOWN (the down-arrow key on the numeric keypad).

4.2.4 Other File-Navigation Functions

The following functions are useful for moving through a file:

Function (and Default Keystrokes)	Description
Pword (CTRL+RIGHT)	Moves the cursor forward (plus) one word
Mword (CTRL+LEFT)	Moves the cursor backward (minus) one word

Ppara	Moves the cursor forward (plus) one paragraph
Mpara	Moves the cursor backward (minus) one paragraph
Mark (CTRL+M)	Defines or moves to a marker, or moves to a specified line number

With the *Mark* function, you can define a marker or move to a marker. Markers constitute a special topic that is discussed in Section 4.4, "Using File Markers."

4.3 Inserting, Copying, and Deleting Text

You may often need to move, copy, or delete blocks of text. The Microsoft Editor is particularly powerful because it provides a variety of ways to define a block of characters.

For example, you can delete a highlighted box, a range of lines, or a stream of text between any two file positions. Sections 4.3.1– 4.3.4 discuss how to work with blocks of text.

4.3.1 Inserting and Deleting Text

Chapter 2, "Edit Now," described how to use the *Paste*, *Insertmode*, and *Delete* functions to insert, move, and delete text.

The following list presents some of the most common commands that use the *Delete* function:

Command (and Default Keystrokes)	Description
Delete (DEL)	Deletes the character at the cursor position. (This command does not join two lines of text, even if the cursor is at the end of the line.)
Arg Delete (ALT+A DEL)	Deletes all text from the cursor position to the end of the line and joins the current line of text with the next line.
Arg cursor-movement Delete (ALT+A *cursor-movement* DEL)	Deletes the highlighted area, whether it is a *linearg*, *streamarg*, or *boxarg*. You can toggle between stream and box selection with the *Boxstream* function (CTRL+B).

The *Delete* function copies all deleted text (except single-character deletions) to the Clipboard. For any argument, the *Meta Delete* function discards the text without copying it to the Clipboard.

To deal with whole lines of text, the Microsoft Editor provides the following functions:

Function (and Default Keystrokes)	Description
Ldelete (CTRL+Y)	Deletes a line of text or a *boxarg*
Linsert (CTRL+N)	Inserts a line of text or a *boxarg*

You can use these functions in commands without an argument or prefix. These functions also take cursor-movement arguments. They produce the same results that *Delete* and *Insert* do, but they always act as if the editor were in box mode.

NOTE *When you want to delete or copy large areas of text, you may find the DIRECTION keys move too slowly. However, you can define a highlighted area with any cursor-movement function that does not take an argument. Mpage (PGUP) and Ppage (PGDN) can be used this way. The Ppara and Mpara functions can also be used this way, but they do not have default key assignments.*

To delete large amounts of text, you can also use a markarg or a numarg with a delete function. See Table A.3 for more information.

4.3.2 Copying Text

To copy text without first deleting it, use the *Copy* function (CTRL+INS), which copies a range of text into the <clipboard> pseudo file. Text in the Clipboard is then inserted into the file when you invoke the *Paste* function. The following list presents different commands that use the *Copy* function:

Command (and Default Keystrokes)	Description
Arg cursor-movement Copy (ALT+A *cursor-movement* CTRL+INS)	Copies the highlighted area into the Clipboard.
Arg numarg Copy (ALT+A *numarg* CTRL+INS)	Copies the specified number of lines into the Clipboard, beginning with the line that the cursor is on.

Arg markarg Copy (ALT+A *markarg* CTRL+INS)	Copies the text between the specified marker and the cursor into the Clipboard. The shape of this region changes depending on whether the editor is in box or stream mode.

The *Paste* function (SHIFT+INS) is useful both for moving and copying text. To move text, first delete it and then invoke *Paste* after moving the cursor to the destination.

NOTE *If you highlight an area on the screen and then invoke Paste, the editor deletes the highlighted area and replaces it with the contents of the Clipboard.*

See Section 4.4 for more information on markers.

4.3.3 Other Insert Commands

The following functions insert specific items at the current cursor position (each function is a complete command). These functions do not have preassigned keys; see Chapter 6, "Function Assignments and Macros," for information on how to assign keys to functions.

Function	Description
Curdate	Inserts current date
Curday	Inserts current day of the week
Curfile	Inserts current file name
Curfileext	Inserts current file extension
Curfilenam	Inserts base name of current file
Curtime	Inserts current time

These functions all use time of execution, rather than time of editor start-up, as the current time.

Although the functions above are not preassigned to any keystrokes, you can assign them to keystrokes by using the technique described in Chapter 6. You can also execute a function by giving its name as input to *Execute* (F7).

For example, the following sequence inserts the date at the cursor position:

1. Invoke *Arg* (press ALT+A)
2. Type the following string: `curdate`
3. Invoke *Execute* (press F7)

4.3.4 Reading a File into the Current File

The *Paste* function can be used in commands that read a file into the current file, as shown below:

Command (and Default Keystrokes)	Description
Arg Arg textarg Paste (ALT+A ALT+A *textarg* SHIFT+INS)	Reads the contents of the file specified by the *textarg* and inserts these contents into the current file. The insertion occurs at the cursor position.
Arg Arg !textarg Paste (ALT+A ALT+A *!textarg* SHIFT+INS)	Reads the output of the system-level command line given as the *textarg*. The output is inserted at the cursor position. For example, if the *textarg* is DIR, then a directory listing is inserted into the file.

4.4 Using File Markers

File markers help you move back and forth through large files. Once you have defined a file marker, you can move quickly to the location marked. You can also use a file marker as input to certain commands. For example, instead of moving the cursor to a marked location, you simply give the name of the marker.

The Microsoft Editor allows you to create any number of file markers. You identify each with a name consisting of alphanumeric characters.

Use the *Mark* function (CTRL+M) to create or go to a marker. The command *Mark* (CTRL+M with no argument) takes you back to the beginning of the file, just as *Arg Mpage* does. The command *Arg Mark* (ALT+A CTRL+M) moves you back to the previous cursor position. This last use of *Mark* is useful for switching back and forth quickly between two locations.

Some of the most powerful uses of the *Mark* function involve commands with arguments, as shown below:

Command (and Default Keystrokes)	Description
Arg numarg Mark (ALT+A *numarg* CTRL+M)	Moves the cursor to the line that you specify. The Microsoft Editor numbers lines beginning with the number 1, so the first line of the file is line 1, the second is line 2, and so forth.
Arg Arg textarg Mark (ALT+A ALT+A *textarg* CTRL+M)	Defines a marker at the current location. This command sets a marker which in turn can be used as input to other functions.
Arg textarg Mark (ALT+A *textarg* CTRL+M)	Moves the cursor directly to a marker you have already defined as a *textarg*.

The marker name may include digits, but must include at least one nondigit character as well.

4.4.1 Functions That Use Markers

The following functions also make use of markers by accepting a previously defined marker name (a *markarg*) as an argument. These functions all use the area in the file defined by the cursor position and the marker. This area, in turn, is interpreted as a stream argument, or *linearg* or *boxarg*, depending on whether the editor is in stream mode or box mode. Recall that the *Boxstream* function (CTRL+B) toggles between these two modes.

Function (and Default Keystrokes)	Description
Assign (ALT+=)	Executes all assignment statements in the defined area (see Chapters 6 and 7 for more information)
Copy (CTRL+INS)	Copies the defined area into the Clipboard
Ldelete (CTRL+Y)	Deletes the defined area
Linsert (CTRL+N)	Fills the defined area with spaces
Qreplace (CTRL+\)	Executes search and replace over the defined area, with query for confirmation
Replace (CTRL+L)	Executes search and replace over the defined area

If you specify a marker the editor cannot find, the editor automatically checks the file listed in the **markfile** switch. See Table A.5 for more information on the **markfile** switch.

4.4.2 Related Functions: Savecur and Restcur

The *Savecur* and *Restcur* functions are similar to *Mark* but do not take arguments. Use *Savecur* to save the current cursor position and *Restcur* to return to that position later. With these two functions, you can save only one position at a time.

No keys are preassigned to *Savecur* or *Restcur*. See Chapter 6, "Function Assignments and Macros," for information on how to assign keys. You can also use *Savecur* and *Restcur* by giving them as input to the *Arg textarg Execute* command, in which *textarg* is the name of the function to execute:

1. Invoke *arg* (press ALT + A)
2. Type *Savecur* or *Restcur*
3. Invoke *Execute* (press F7)

4.5 Searching and Replacing

The *Psearch* function (F3) directs the editor to conduct a forward search (a "plus search") for the next occurrence of the specified string. All searches take place from the current cursor position to the end of the file.

The most common uses of *Psearch* consist of the following commands:

Command (and Default Keystrokes)	Description
Arg textarg Psearch (ALT+A *textarg* F3)	Directs the editor to look for the string given as *textarg*. The editor scrolls the screen, if necessary, and moves the cursor to the next occurrence of *textarg* in the file.
Psearch (F3)	Directs the editor to look for the most recently specified search string.

Arg Psearch
(ALT+A F3)

Directs the editor to take the word at the current cursor position as the search string. (In other words, the search string consists of all characters from the cursor to the first blank or new line.)

You can search backward with *Msearch* (a "minus search"). The *Msearch* function (F4) uses syntax identical to *Psearch*. Backward searches take place from the current cursor position to the beginning of the file. In addition, *Msearch* with no argument assumes the same search string that was specified with *Psearch*. Therefore, after searching forward for a string, you can search backward for the same string just by pressing F4.

All versions of the *Psearch* and *Msearch* commands are affected by case sensitivity. By default, case sensitivity is off, so the editor carries out searches for strings without distinguishing between uppercase and lowercase letters. However, you can change this behavior by turning the **case** switch on, using the syntax for switch settings explained in Chapter 7, "Switches, Assignments, and the TOOLS.INI File."

You can also temporarily reverse the setting of the **case** switch by turning on the *Meta* prefix. Therefore, if case sensitivity is off, you can conduct a case-sensitive search with the *Meta Psearch* (F9 F3) or *Meta Msearch* (F9 F4) command.

4.5.1 Searching for a Pattern of Text

The commands described above search for an exact match of the string you specify. However, sometimes you may want to search for a set of different strings: for example, any word that begins with "B" and ends with "ing."

You can search for a pattern of text by specifying a "regular expression." A regular expression is a string that specifies a pattern of text by using certain special characters. Chapter 5 describes how to specify regular expressions.

The command *Arg Arg textarg Psearch* (ALT+A ALT+A *textarg* F3) searches forward for a string that matches the regular expression specified as the *textarg*. The command *Arg Arg textarg Msearch* (ALT+A ALT+A *textarg* F4) searches backward for a string that matches the regular expression specified as the *textarg*.

Regular-expression searches are affected by case sensitivity, as explained in the previous section.

4.5.2 Searching the File Globally

The *Searchall* function (SHIFT+F6) takes the same syntax that *Psearch* does, but instead of finding the next occurrence of a string, the *Searchall* function highlights every string in the file that matches the search string. For example, to highlight every occurrence of the word `float` in a file, follow these steps:

1. Invoke *Arg* (press ALT+A)
2. Type the following search string: `float`
3. Invoke *Searchall* by pressing SHIFT+F6

Any action other than cursor movement removes the highlight. The color of the highlight is normally different from the highlight color in cursor-movement arguments.

4.5.3 Searching a Series of Files

You can search a series of files without leaving the editor. The *Mgrep* command takes the same syntax that *Psearch* and *Msearch* do. For example, the command *Arg Arg textarg Mgrep* searches for a regular expression. The editor responds to *Mgrep* by placing all strings found in the <compile> pseudo file. You can look at the file by using *Setfile* or invoking the *Nextmsg* function.

Before using *Mgrep*, place the list of files to search in a macro named `mgreplist`. This list can contain DOS wildcards and environment variables. For example, the following steps direct *Mgrep* to search the following files: the file JUNK.TXT; files that have a .C extension and are in the current directory; and files that have a .H extension and are in any directory listed in the INCLUDE environment varible:

1. Invoke *Arg* (press ALT+A)
2. Define the value of `mgreplist` by typing the following:

```
mgreplist:="JUNK.TXT *.C $INCLUDE:*.H"
```

3. Invoke *Assign* by pressing ALT+= (hold down the ALT key and press the equals sign)

You refer to an environment variable with the following syntax, in which *ENVAR* is the name of the variable. The name must be entered in uppercase characters:

$*ENVAR*:

4.5.4 Search-and-Replace Functions

To replace repeated occurrences of one text string by another, use the search-and-replace function *Replace* (CTRL+L). By default, the replacement happens from the cursor position to the end of the file. However, as described below, you can restrict the range over which the replacement happens.

No matter what command syntax you use with *Replace*, the editor reacts by prompting you for a search string and a replacement string, and then executing the search and replace. If you have used *Replace* or *Qreplace* ("query replace") before, the previous value of the search or replace string appears on the dialog line. To use the string displayed, press ENTER.

The identical commands *Replace* and *Arg Replace* execute replacement from the current cursor position to the end of the file. You can also specify a range for the replacement by using one of the following commands:

Command	Default Keystrokes
Arg linearg Replace	ALT+A *linearg* CTRL+L
Arg numarg Replace	ALT+A *numarg* CTRL+L
Arg boxarg Replace	ALT+A *boxarg* CTRL+L
Arg streamarg Replace	ALT+A *streamarg* CTRL+L
Arg markarg Replace	ALT+A *markarg* CTRL+L

If you specify a *numarg*, the replacement operation is limited to the specified number of lines, beginning with the current line. If you specify a *linearg*, *streamarg*, or *boxarg*, the replacement occurs only within the highlighted area. If you specify a *markarg*, the replacement occurs in the region of text between the cursor position and the marker. The shape of the region changes depending on whether the editor is in box mode or stream mode.

The *Replace* function is most efficient when you are sure you want the replacement to be executed in every case. If you want to regulate how often the replacement occurs, use *Qreplace* (CTRL+\). This function takes the same syntax as *Replace*, but prompts you for confirmation before each replacement. *Qreplace* asks you to press Y for yes, N for no, or A for all, which causes replacement to proceed without further confirmation. Pressing Q (quit) terminates replacement.

The *Replace* and *Qreplace* functions both take regular expressions as search strings when you introduce the argument with *Arg Arg* instead of *Arg*. (See Chapter 5 for information on regular expressions.) Otherwise, syntax is identical, and the functions accept the same arguments.

Search and replacing is affected by case sensitivity, as explained in Section 4.5.1.

You can use the *Mreplace* function to execute replacements throughout a series of files. See Appendix A, "Reference Tables," for more information.

4.6 Compiling

One of the strengths of the Microsoft Editor is its capability as a development environment. You can write a program and compile (or assemble) from within the editor. If the compilation fails, you can make corrections to the source file when you view the errors and then compile again.

Ordinarily, a compiler sends error messages directly to the screen while you are outside the editor. When you compile from within the Microsoft Editor, however, the error messages are displayed on the dialog line. The *Nextmsg* function (SHIFT+F3) displays the error messages in sequence and positions the cursor at the beginning of the line with the next error. You can make corrections immediately.

The *Compile* function (CTRL+F3) appears in a variety of commands, as shown in Section 4.6.1.

4.6.1 Invoking Compilers and Other Utilities

When you run the protected-mode version of the editor under OS/2, compilations run in the background and the editor beeps when the compilation is completed. (While a background compilation is running, the letters `BP` appear on the status line.) When running the real-mode version of the editor (under DOS or the OS/2 3.*x* compatibility box), you cannot edit again until the compilation has completed.

With the Microsoft Editor's compilation capability, you can invoke any program or utility and specify any command-line options. To invoke a program directly, use one of the following commands:

Command (and Default Keystrokes)	**Description**
Arg Arg textarg Compile (ALT+A ALT+A *textarg* CTRL+F3)	Runs the system-level command described by *textarg*. Typically, this command runs the compiler.
Arg Compile (ALT+A CTRL+F3)	Runs a compilation according to the **extmake** switch.
Arg textarg Compile (ALT+A CTRL+F3)	Runs a compilation according to the "text" setting of the **extmake** switch. This version of the command is typically used with the NMAKE utility.

Arg Meta Compile (ALT+A F9 CTRL+F3)	Kills any compilation running in the background after prompting for confirmation (OS/2 only).

Usually, it is more convenient to set your compile command once by setting the **extmake** switch and giving the *Arg Compile* command each time you compile.

A "switch" is a variable that you can set to control the editor's behavior. See Chapter 7, "Switches, Assignments, and the TOOLS.INI File," for more information on switches and how to set them.

The *Arg Compile* command examines the file extension of the current file and executes the corresponding **extmake** setting. The **extmake** switch can have a different setting for each file extension. The general form of an **extmake** assignment is

extmake:*ext command-line*

in which *ext* is a file extension and *command-line* is the compile command to invoke for files with this extension. For example, suppose you want to invoke the following compile command for use with .C files:

```
cl /AL /Zi /Ox %s
```

The characters `%s` represent the name of the current file. To automatically invoke this compile command with *Arg Compile*, you would first follow these steps to set the **extmake** switch:

1. Invoke *Arg* (press ALT+A)
2. Type the following text: `extmake:c cl /AL /Zi /Ox %s`
3. Press ALT+= to invoke *Assign* (hold down the ALT key and press the equals sign)

The editor then invokes the desired command line whenever you give the *Arg Compile* command and are editing a .C file.

To use the *Arg textarg Compile* command, first set **extmake** with a **text** extension:

extmake:text *command-line*

If the symbol `%s` appears in *command-line*, it is replaced with the text argument to *Compile*. For example, if **extmake** has the setting

```
extmake:text nmake %s
```

and you pass the text argument `projectx` to the *Compile* function, the editor executes the following system-level command:

```
nmake projectx
```

4.6.2 Viewing Error Output

To generate error output that you can view from within the editor, the compiler or assembler must output errors in one of the following formats:

filename row column: *message*
filename (*row*, *column*): *message*
filename (*row*): *message*
filename: *row*: *message*
"*filename*", *row column*: *message*

The Microsoft Editor, in turn, reads the error output directly and responds by moving the cursor to each location where an error was reported while displaying the *message* on the dialog line. (The method for moving between error locations is described below.) The following programs output error messages in a format readable by the Microsoft Editor:

- Microsoft C Optimizing Compiler
- Microsoft Macro Assembler
- Microsoft Pascal Compiler 4.0
- Microsoft BASIC Compiler 6.0
- Microsoft FORTRAN Optimizing Compiler 4.1

NOTE *With the Pascal and BASIC compilers, you must use the /Z command-line option with either the PL or BC driver to generate error output that the Microsoft Editor can read. (The* ***extmake*** *switch, discussed in Chapter 7, uses the /Z option by default.)*

When a compilation fails in real mode, the editor displays the first error message and positions the cursor at the line with the error. The *Nextmsg* function (SHIFT+F3) displays the next error message and repositions the cursor at the appropriate line.

When a compilation fails in protected mode, the editor beeps rather than interrupts your current activity. If you had previously viewed the results of an earlier compilation, give the *Arg Meta Nextmsg* command (ALT+A F9 SHIFT+F3) to advance to the current set of error messages. Then, regardless of how many compilations you are running, press SHIFT+F3 to display the first error message and move the cursor to the line with that error.

Some common ways to use the *Nextmsg* function are shown below:

Command (and Default Keystrokes)	Description
Nextmsg (SHIFT+F3)	Moves cursor to location of next error message and displays text of error message on the dialog line.
Arg numarg Nextmsg (ALT+A *numarg* SHIFT+F3)	Moves forward or backward by *numarg* error messages. For example, if *numarg* is `-1`, moves the cursor to the previous error message.
Arg Meta Nextmsg (ALT+A A SHIFT+F3)	OS/2 only. Advances to next set of error messages. Under OS/2, the editor maintains error messages for all compilations. This command directs the editor to advance to the error messages for the subsequent compilation. This feature supports simultaneous background compilations. No matter how many compilations you executed, you can view every set of error messages.

4.6.3 Viewing the Dynamic-Compile Log

Reviewing error messages as described in the previous section is useful when each error message corresponds to a location in your source file. However, some messages (such as linker errors) do not correspond to specific lines of code. To review these errors, as well as general compilation errors, you may want to view the actual error-message text.

The editor keeps the complete error output of each compilation—including output from any utilities that were invoked—in the <compile> pseudo file. If the editor is running in real mode, you can view the <compile> pseudo file after a compilation is complete by following these steps:

1. Invoke *Arg* (press ALT+A)
2. Type the following text: `<compile>`
3. Invoke *Setfile* (press F2)

If the editor is running in OS/2 protected mode, you can view the error output in the same manner described above. You can also view the <compile> pseudo file as the compilation is running. Under protected mode, the <compile> pseudo file is updated dynamically; as each error message is produced by the compiler or utility, the editor adds the message to the end of this file.

Using the techniques described in the next section, you can open <compile> as a separate window. This lets you continue to edit your source file while watching error messages appear the instant an error is detected. The command *Arg Meta Compile* kills the compilation.

4.7 Using Editing Windows

An "editing window" is a division of the screen that functions independently from other portions of the screen. When you have two or more windows present, each functions as a miniature screen. For example, one window can view lines 5–15 while another window views lines 90–97. You can even use windows to view two or more files simultaneously.

Although windows are tiled, they can view overlapping areas of text. Changes and highlighting are reflected simultaneously in all windows that view the area of altered text.

You can have up to eight windows on the screen and create either horizontal or vertical divisions between windows. The command *Window* (F6 with no arguments) moves the cursor between windows.To create or merge a window, move the cursor to the row or column at which you want to create a new division, and give one of the following commands:

Command (and Default Keystrokes)	**Description**
Arg Window (ALT+A F6)	Creates a horizontal window (split at the cursor row)
Arg Arg Window (ALT+A ALT+A F6)	Creates a vertical window (split at the cursor column)
Meta Window (F9 F6)	Closes the current window by merging it with the window to the right or below

Each window must have a minimum of 5 lines and 10 columns. If you try to create a window of a smaller size, the command fails.

4.8 Working with Multiple Files

You can load a new file in the screen or current window with the *Setfile* function.

Command (and Default Keystrokes)	Description
Arg textarg Setfile (ALT+A *textarg* F2)	Loads the file specified in the *textarg*.
Setfile (F2)	Loads the previous file. You can use *Setfile* to move back and forth between two files.

You can also use *Setfile* by following these steps:

1. Bring up the information file with the *Information* function (press F10).
2. Move the cursor to the beginning of the name of a file.
3. Select the file that the cursor is on by giving the command *Arg Setfile* (press ALT+A F2).

The information file contains the names of all files that you have edited before, up to the limit specified by the **tmpsav** switch. (See Table A.5 for more information on switches.) Active files—files that have been edited during this session—are listed with their current lengths. This file also tells whether the text in the Clipboard was copied in line (box) or stream mode.

When an old file is reloaded, the editor remembers cursor and window information from the last time you edited the file. The editor stores this information in the file M.TMP (or MEP.TMP, if the editor is named MEP).

The *Arg textarg Setfile* command accepts wild-card characters (**?** matches any character and ***** matches any string) in the *textarg*. The command responds by displaying a list of files that match the *textarg*. You can then select a file by using the steps outlined above. For example, the following sequence causes the editor to list all files with a .C extension:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following: `*.c`
3. Invoke the *Setfile* function (press F2)

4.9 Printing a File

You can print a file without leaving the Microsoft Editor. You can also print a highlighted area or a series of files. Use the *Print* function (CTRL+F8) to perform each of these actions:

Command (and Default Keystrokes)	Description
Print (CTRL+F8)	Prints all of the current file.
Arg cursor-movement Print (ALT+A *cursor-movement* CTRL+F8)	Prints the highlighted area.
Arg textarg Print (ALT+A *textarg* CTRL+F8)	Prints the file or files specified in *textarg*. If there is more than one file, separate them with a space.

By default, the *Print* function responds by sending the specified output to the LPT1 device. However, you can specify a different print command by setting the **printcmd** switch.

For example, suppose you want to print a file by using the following command:

```
COPY %s LPT2
```

The characters `%s` represent the file name (in the case of highlighted regions, the editor creates a temporary file). To invoke this command each time you use the *Print* function, you first follow these steps to set the **printcmd** switch:

1. Invoke *Arg* (press ALT+A)
2. Type the following text: `printcmd:COPY %s LPT2`
3. Press ALT+= to invoke *Assign* (hold down the ALT key and type the equals sign)

When printing from within the protected-mode editor under OS/2, the printing occurs as a background operation and `BP` is displayed at the lower-right-hand corner of the screen. The <print> pseudo file maintains a log of the printed output.

NOTE *A "switch" is a variable you can set to control the editor's behavior. See Chapter 7, "Switches, Assignments, and the TOOLS.INI File," for more information on switches and how to set them.*

CHAPTER 5

Regular Expressions

A "regular expression" is a special search string that matches a *pattern* of text rather than a specific sequence of characters. With regular expressions, you can search for such targets as every five-digit number, or every string in quotes, without having to specify the exact text to search for.

In a regular expression, certain characters lose their literal meaning, becoming symbols or placeholders that specify the text pattern you want to match. In a regular expression such as `a[123]`, the `a` is still the literal character "a". The brackets, however, are not search targets. Rather, they enclose a set of characters, any one of which is a match. Therefore, the regular expression `a[123]` matches any of these strings:

```
a1
a2
a3
```

The Microsoft Editor supports two versions of regular-expression syntax: UNIX® and M 1.0. The UNIX syntax provides compatibility with the syntax used by programming utilities for the UNIX and XENIX® operating systems and CodeView®. M 1.0 syntax does not provide this compatibility; however, it offers more power.

This chapter examines three aspects of regular-expression syntax:

- Choosing the syntax
- Using UNIX syntax
- Using M 1.0 syntax

You can use regular expressions with the search functions (*Psearch*, *Msearch*, *Replace*, and *Qreplace*). Each of these functions recognizes a regular expression (rather than an ordinary text string) when you use *Arg Arg* to introduce the string.

5.1 Choosing the Syntax

The Microsoft Editor supports two forms of regular-expression syntax: the M 1.0 form and the UNIX standard. If you know the UNIX standard, you can use it immediately. However, the slightly different M 1.0 syntax of the Microsoft Editor is more powerful.

By default, the editor recognizes the UNIX syntax for regular expressions. To use the M 1.0 syntax instead, follow these steps:

1. Invoke *Arg* (press ALT+A)
2. Type the following: `unixre:no`
3. Invoke *Assign* (press ALT+=)

To enable the standard UNIX syntax, follow the same steps but type `unixre:` instead of `unixre:no`.

If you would like the editor to run automatically in M 1.0 mode, rather than having to manually invoke the *Assign* function, add a `unixre:no` entry to the TOOLS.INI file. See Chapter 7, "Switches, Assignments, and the TOOLS.INI File," for more information.

5.2 UNIX® Regular-Expression Syntax

The UNIX regular-expression syntax is compatibile with UNIX utilities and CodeView. By default, the editor uses UNIX syntax for regular expressions. The M 1.0 syntax offers the same capabilities as the UNIX syntax, plus a few additional features.

5.2.1 UNIX Regular Expressions as Simple Strings

The power of regular expressions comes from the use of the special characters and character sequences listed below. A regular expression that does not contain these special characters or character sequences acts as a literal text string:

```
\( \) [ ] ! . ^ $ *
```

For example, the regular expression `match me precisely` matches only a literal occurrence of itself because it contains no special characters.

5.2.2 UNIX Special Characters

The UNIX operators offer the standard pattern-matching capabilities found in many other editors and utilities that use regular expressions.

The list below describes the special characters that have a simple usage. The term *class* has a special meaning defined below. All other characters should be interpreted literally.

Expression	Description
\	Escape. Causes the editor to ignore the special meaning of the next character. For example, the expression `\.` matches `.` in the text file; the expression `\^` matches `^`; and the expression `\\` matches `\`.
.	Wildcard. Matches any single character. For example, the expression `a.a` matches `aaa`, `aBa`, and `a1a`, but not `aBBBa`.
^	Beginning of line. For example, `^The` matches the word `The` only when it occurs at the beginning of a line.
$	End of line. For example, `end$` matches the word `end` only when it occurs at the end of a line.
[*class*]	Character class. Matches any one character in the class. Use a dash (-) to specify a contiguous range of ASCII values. For example, `[a-zA-Z0-9]` matches any letter or digit, and `[abc]` matches `a`, `b`, or `c`.
[^*class*]	Inverse of character class. Matches any character not specified in the class.

The special characters with more complex usage are described in the following list. The expression *X* is a placeholder representing a regular expression that is either a single character, a group of characters enclosed in brackets ([ ]), or the regular-expression delimiters \(and \). The letter *n* represents a one-digit number.

Expression	Description
*X**	Repeat operator. Matches zero or more occurrences of *X*. For example, the regular expression `ba*b` matches `baaab`, `bab`, and `bb`. This operator always matches as many characters as possible.
\(... \)	Tagged expression. A marked substring which you can refer to elsewhere in the search string, or in a replacement string, as *\n*. When a tagged expression is referred to in a search string, the editor finds text with the tagged expression repeated. When a tagged expression is referred to in a replacement string, the editor reuses part of the text it is replacing. The exact use of tags is explained in Sections 5.2.4 and 5.2.5. Characters falling between \(and \) are treated as a group.
\n	Reference to the characters matched by a tagged expression. The number *n* indicates which expression. The first tagged expression is represented as \1, the second as \2, and so on.

The procedure below uses some of the special characters presented in this section. To find the next occurrence of a number (that is, a string of digits) beginning with a digit 1 or 2, perform the following sequence of keystrokes:

1. Invoke *Arg* twice (press ALT+A twice)
2. Type the following characters: `[12][0-9]*`
3. Invoke *Psearch* (press F3)

5.2.3 Combining UNIX Special Characters

Special characters are most powerful when used in combination. For example, the wildcard (`.`) and repeat (`*`) characters are often used together:

```
.*
```

The expression above means "match any string of characters." Although this expression is not useful by itself, it is quite useful when part of a larger expression. For example,

```
B.*ing
```

means "match any string beginning with `B` and ending with `ing`".

In cases where a single character is surrounded by two operators, regular expressions are interpreted from left to right. For example, suppose the following is part of a regular expression:

```
\1*
```

In the expression above, the repeat operator (*) applies to the characters \1 as a single unit.

If you want the repeat operator to apply to a group, enclose the group inside the symbols \(and \). These characters tag an expression (as explained in the next section) and are useful for treating a series of characters as a group.

5.2.4 Tagged Expressions in the UNIX Search String

A "tagged expression" is a substring delimited by the symbols \(and \). You can enter any regular-expression characters between these delimiters. Tagged expressions are used to specify text patterns that contain repeated elements and to mark a string for reuse.

The editor first searches for a character string that matches the entire regular expression. It then tags each substring specified in a tagged expression. Up to nine substrings at a time may be tagged.

Once a tagged expression has been matched, you can refer to the specific string of characters that matched that expression. Use the syntax

\n

in which *n* is a number that selects the expression. The symbol \1 represents the first tagged expression, the symbol \2 represents the second tagged expression, and so on. The use of \n does not search for a new match for the tagged expression. Rather, it matches only an occurrence of the same characters that the tagged expression itself matched.

For example, consider the following expression:

```
\(.\)\1\1
```

The expression above means, "match any character, then see if it's followed by two occurrences of the same character." The following strings all satisfy this requirement:

```
aaa
XXX
111
```

Note that this regular expression is not equivalent to . . . (three wild cards). The expression . . . matches any three characters; the characters do not need to be the same.

The next expression is more complex:

```
\([A-Za-z]*\)==\1
```

This expression means "match any number of letters, then see if the letters are followed by two equals signs (==) and a repetition of the original group." This expression matches the first two strings below but not the third:

```
ABCxyz==ABCxyz
i==i
ABCxyz==KBCxjj
```

5.2.5 Tagged Expressions in the UNIX Replacement String

You can refer to tagged expressions in replacement strings as well as in search strings. Parts of the string to be replaced may be reused by referring to the tagged expressions that originally matched those parts. Use the syntax described in the previous section.

For example, suppose you want to find all occurrences of *hexdigits***H** and replace them with strings of the form **16#***hexdigits*. You can search for strings of the form *hexdigits***H** by specifying the regular expression

```
\([0-9a-fA-F]*\)H
```

and then specifying the following replacement string:

```
16#\1
```

The result is that the Microsoft Editor searches for any occurrence of one or more hexadecimal digits (digits 0–9 and the letters a–f) followed by the letter **H**. Each matching string is replaced by a new string that consists of the original digits (which were tagged so they could be reused) and the prefix **16#**. For example, the string `1a000H` is replaced with the string `16#1a000`.

Use two backslashes (`\\`) to represent a literal backslash (`\`). Within replacement strings, all characters except the backslashes are literals. The backslash is considered to be the first character of a regular-expression reference, such as `\4`.

5.3 M 1.0 Regular-Expression Syntax

The M 1.0 syntax is fully compatible with the regular-expression syntax used by Version 1.0 of the Microsoft Editor. This syntax offers all the features of UNIX regular-expression syntax (though sometimes using different characters), plus additional features.

To choose M 1.0 syntax, you must set the **unixre** switch to off, as described in Section 5.1, "Choosing the Syntax."

5.3.1 M 1.0 Regular Expressions as Simple Strings

The power of regular expressions comes from the use of the special characters listed below. A regular expression that does not contain these special characters acts as if it were a literal text string:

\ { } ( ) [ ] ! ~ : ? ^ $ + * @ #

For example, the regular expression `match me precisely` matches only a literal occurrence of itself because it contains no special characters.

5.3.2 M 1.0 Special Characters

The M 1.0 syntax offers a rich set of pattern-matching capabilities. Most of the special characters described below have analogs in other editors and utilities that use regular expressions.

The list below describes the special characters that have a simple usage. The term *class* has a special meaning defined below. All other characters should be interpreted literally.

Expression	Description
\	Escape. Causes the editor to ignore the special meaning of the next character. For example, the expression `\?` matches `?` in the text file; the expression `\^` matches `^`; and the expression `\\` matches `\`.
?	Wildcard. Matches any single character. For example, the expression `a?a` matches `aaa`, `aBa`, and `a1a`, but not `aBBBa`.
^	Beginning of line. For example, `^The` matches the word `The` only when it occurs at the beginning of a line.
$	End of line. For example, `end$` matches the word `end` only when it occurs at the end of a line.
[*class*]	Character class. Matches any one character in the class. Use a dash (-) to specify ranges. For example, `[a-zA-Z0-9]` matches any character or digit, and `[abc]` matches `a`, `b`, or `c`.
[~*class*]	Noncharacter class. Matches any character not specified in the class.

The special characters with more complex usage are described in the following list. The expression *X* is a placeholder representing a regular expression that is either a single character or a group of characters enclosed in parentheses (( )), brackets ([ ]), or braces ({ }). The placeholders *X1*, *X2*, and so on, represent any regular expression.

Expression	Description
*X**	Minimal matching. Matches zero or more occurrences of *X*. For example: the regular expression `ba*b` matches `baaab`, `bab`, and `bb`.
X+	Minimal matching plus (shorthand for *XX**). Matches one or more occurrences of *X*. The regular expression `ba+b` matches `baab` and `bab` but not `bb`.
X@	Maximal matching. Identical to *X**, except for differences in matching method explained in Section 5.3.4.
X#	Maximal matching plus. Identical to *X+*, except for differences in matching method explained in Section 5.3.4.
(X1!X2!...!Xn)	Alternation. Matches either *X1*, *X2*, and so forth. It tries to match them in that order and switches from *Xi* to *Xi+1* only if the rest of the expression fails to match. For example, the regular expression `(ww!xx!xxyy)zz` matches `xxzz` on the second alternative and `xxyyzz` on the third.
~X	Not function. Matches nothing, but checks to see if the string matches *X* at this point and fails if it does. For example, `^~(if!while)?*$` matches all lines that do not begin with `if` or `while`.
X^n	Power function. Matches exactly *n* copies of *X*. For example, `w^4` matches `wwww` and `(a?)^3` matches `a#aba5`.

{...}	Tagged expression, which is a string of characters you identify so that you can refer to them elsewhere, as $*n*. By referring to a tagged expression in a search string, you cause the editor to look for patterns involving duplication. By referring to a tagged expression in a replacement string, you cause the editor to reuse part of the text that it is replacing. The exact use of tags is explained in Sections 5.3.5 and 5.3.6. Characters within braces are treated as a group.
$*n*	Reference to a previously tagged substring. The number *n* indicates which substring. The first tagged substring is represented as $1, the second as $2, and so on. $0 represents the entire matched string.
:*letter*	Predefined string. The list of predefined strings is given in Section 5.3.7.

The procedure below uses some of the special characters presented in this section. To find the next occurrence of a number (that is, a string of digits) beginning with a digit 1 or 2, perform the following sequence of keystrokes:

1. Invoke *Arg* twice (press ALT + A twice)
2. Type the following characters: `[12][0-9]*`
3. Invoke *Psearch* (press F3)

5.3.3 Combining M 1.0 Special Characters

Special characters are most powerful when used in combination. For example, the `?` and `*` characters are often used together:

```
?*
```

The expression above means "match any string of characters." Although this expression is not useful by itself, it is quite useful when part of a larger expression. For example,

```
B?*ing
```

means "match any characters beginning with `B` and ending with `ing`".

Many of the special characters in the previous section (such as + and *) are *operators*; they work with other characters to form expressions. These operators usually apply to the previous character or an expression enclosed in braces, brackets, or parentheses. However, it is possible to find exceptions to this rule. For example, the following expression is meaningful:

```
S^2+
```

The plus sign (+) applies to the entire expression `S^2`. In effect, this expression means, "match any even number of occurrences of the letter S." Therefore, this expression means the same as this:

```
(S^2)+
```

Precedence is left to right in cases where a single character is surrounded by two operators. If there is any doubt about the precedence of operators, use parentheses.

5.3.4 M 1.0 Matching Method

The matching method you use is significant only when you use a search-and-replace function. The term "matching method" refers to the technique used to match repeated expressions. For example, does `a*` match as few or as many characters as it can? The answer depends on the matching method. There are two matching methods:

Method	Description
Minimal	The minimal method matches as few characters as possible in order to find a match. For example, `a+` matches only the first character in `aaaaaa`. However, `ba+b` matches the entire string `baaaaaab`, since it is necessary to match every occurrence of `a` in order to match both occurrences of `b`.
Maximal	The maximal method always matches as many characters as it can. For example, `a#` matches the entire string `aaaaaa`.

The significance of these two methods may not be apparent until you use search and replace. For example, if `a+` (minimal matching plus) is the search string and `EE` is the replacement string,

```
aaaaa
```

is replaced with

```
EEEEEEEEEE
```

because each occurrence of `a` is immediately replaced by `EE`. However, if `a#` (maximal matching plus) is the search string, the same string is replaced with

```
EE
```

because the entire string `aaaaa` is matched at once and replaced with `EE`.

5.3.5 Tagged Expressions in the M 1.0 Search String

A "tagged expression" is a substring delimited by curly braces ({ }). You can enter any regular-expression characters between these delimiters. Tagged expressions are used to specify text patterns that contain repeated elements and to mark a string for reuse.

The editor first searches for a character string that matches the entire regular expression. It then tags each substring specified in a tagged expression. Up to nine substrings at a time may be tagged.

Once an expression is tagged, you can refer to the specific string of characters that matched the expression. Use the syntax

$*n*

in which *n* is a number that selects the expression. The symbol $1 represents the first tagged expression, the symbol $2 represents the second tagged expression, and so on. The use of **$***n* does not search for a new match for the tagged expression. Rather, it matches only an occurrence of the same characters that the tagged expression itself matched.

For example, consider the following expression:

```
{?}$1$1
```

The expression above means, "match any character, then see if it's followed by two occurrences of the same character." The following strings all satisfy this requirement:

```
aaa
XXX
111
```

Note that this regular expression is not equivalent to `???` (three wild cards). The expression `???` matches any three characters; the characters do not need to be the same.

The next expression is more complex:

```
{[A-Za-z]*}==$1
```

This expression means "match any number of letters, then see if the letters are followed by two equals signs (==) and a repetition of the original group." This expression matches the first two strings below but not the third:

```
ABCxyz==ABCxyz
i==i
ABCxyz==KBCxjj
```

5.3.6 Tagged Expressions in the M 1.0 Replacement String

You can refer to tagged expressions in replacement strings as well as in search strings. Parts of the string to be replaced may be reused by referring to the tagged expressions that originally matched those parts. Use the syntax described in the previous section.

For example, suppose you want to find all occurrences of *hexdigits***H** and replace them with strings of the form **16#***hexdigits*. You can search for strings of the form *hexdigits***H** by specifying the regular expression

```
{[0-9a-fA-F]+}H
```

and then specifying the following replacement string:

```
16#$1
```

The result is that the Microsoft Editor searches for any occurrence of one or more hexadecimal digits (digits 0–9 and the letters a–f) followed by the letter **H**. Each matching string is replaced by a new string that consists of the original digits (which were tagged so they could be reused) and the prefix **16#**. For example, the string `1a000H` is replaced with the string `16#1a000`.

The editor recognizes six special characters—**$ (,) - **—within replacement strings. Each of these characters should be preceded by a backslash (\ ) if you want to use a literal occurrence of one of them in a replacement string. The # sign in the replacement string shown above is not treated as a regular-expression character.

You can also use the expression

$(*w*, *n*)

in which *w* is a field length. This number can be positive or negative. A negative number indicates left justification.

If *w* is greater than the length of the tagged expression, the editor right justifies the tagged expression within the field and pads the field with leading spaces. If *w* is greater than the length of the expression but is preceded by a negative sign, the editor left justifies the expression and pads the field with trailing spaces. If *w* is equal or less than the length of the expression, the editor prints the whole expression but does not pad with spaces.

5.3.7 Predefined M 1.0 Regular Expressions

Several M 1.0 regular expressions are defined in Table 5.1 for your convenience. You can use them by entering **:** *letter* in a regular expression.

Table 5.1 Predefined Expressions

Letter	Meaning	Description
:a	[a–zA–Z0–9]	Alphanumeric
:b	([\t]#)	White space
:c	[a–zA–Z]	Alphabetic
:d	[0–9]	Digit
:f	([~"\[\]\:<|>+=;,\\/]#)	Portion of a file name
:h	([0–9a–fA–F]#)	Hexadecimal number
:i	([a–zA–Z_$][a–zA–Z0–9_$]@)	C-language identifier
:n	([0–9]#[0–9]@![0–9]@.[0–9]#![0–9]#)	Number
:p	(([a–z]\:!)(\\!)(:f(:f !)\\)@:f (.:f !))	Path
:q	("[~"]@"!'[~']@')	Quoted string
:w	([a–zA–Z]#)	Word
:z	([0–9]#)	Integer

CHAPTER 6

6

Function Assignments and Macros

One of the strengths of the Microsoft Editor is you can customize it to your own needs. You can change screen characteristics, select default behavior, and specify which functions are connected to which keystrokes. You can also create new editing functions. The editor supports four techniques for customization, described in Section 6.1.

This chapter discusses two of those techniques: function assignments and macros. Function assignments alter the action of keystrokes. Macros are editing commands that you create using a simple syntax. The other two methods for customizing the editor, switch settings and C extensions, are described in Chapters 7, 8, and 9.

This chapter covers the following topics:

- Techniques for customizing the editor
- Assigning functions to keystrokes
- Creating macros

6.1 The Four Techniques for Customizing the Editor

Each of the four techniques for customizing the editor has a distinct purpose. Yet you can use these techniques in combination. For example, an extension can execute a macro, and a macro can make an assignment. Each technique is described below:

Technique	Description
Function assignment	Assigns a function to a particular keystroke. This capability lets you control the meaning of all recognized keystrokes.
Macro assignment	Creates a new editing command out of strings of text and existing editing functions. Macros use a simple syntax and can be created very fast. Use macros when you need to quickly define a new command or repetitive activity.
Switch assignment	Alters a specific editor condition, for example, screen colors, screen height, scrolling behavior, tab behavior, and many other conditions. These conditions are adjusted by setting various editing "switches," as explained in Chapter 7.
C extension	Generates a new editing function compiled with Microsoft C or assembled with the Macro Assembler. Extensions take longer to create than macros, but they run faster and are more flexible. See Chapter 8 for more information.

6.2 Assigning Functions to Keystrokes

A function assignment lets you alter the meaning of any keystroke. In this context, a "keystroke" is any recognized function key, special key, or ALT+, SHIFT+, or CTRL+key combination. You can also alter the meaning of alphanumeric keys, but doing so can interfere with your ability to type characters.

NOTE *Any editing function, macro, or extension function can be assigned to a keystroke as described in this chapter.*

When a new assignment has been made, you can use that keystroke to invoke the function at any time during the editing session. Take into account the following points when assigning functions to keystrokes:

1. The function assignments you make during the editing session are lost when you exit the editor. See Chapter 7, "Switches, Assignments, and the TOOLS.INI File," for information on making assignments that are automatically recognized in each editing session.

When using the assignments screen to view and alter assignments, you have the option of automatically saving your changes. See Section 6.2.2, "Viewing and Changing Function Assignments," for more information.

2. A function can be assigned to more than one keystroke at the same time.
3. Each keystroke can have only one function assigned to it at any one time. Therefore, a new function assignment to a given keystroke cancels any previous meaning the keystroke may have had. Assigning the *Unassigned* function, as explained in Section 6.2.3, cancels the keystroke's previous meaning without substituting a new meaning.

6.2.1 Making Function Assignments

There are two ways to make a function assignment. You can use the *Assign* function, or edit the assignments screen as described in the next section.

To assign a function to a keystroke with the *Assign* command, issue the *Arg textarg Assign* command (ALT+A *textarg* ALT+=), where *textarg* uses the following syntax:

functionname:*keystroke*

Here, *keystroke* may be any of the following:

1. Numeric keys: 0 through 9
2. Lowercase letter keys: a through z
3. Uppercase letter keys: A through Z
4. Function keys: F1 through F10 (F11 and F12 recognized for enhanced keyboards)
5. Lowercase punctuation: ` - = [ ] \ ; ' , . /
6. Uppercase punctuation: ~ ! @ # $ % ^ & * ( ) _ + { } | : " ?
7. Numeric-keypad white keys when NUMLOCK is turned off: HOME, END, LEFT, RIGHT, UP, DOWN, PGUP, PGDN, INS, and GOTO, which corresponds to the numeric-keypad 5 key.
8. Numeric-keypad white keys when NUMLOCK is turned on: 0 through 9. To assign a function to the 4 key on the numeric keypad, enter the following as the *keystroke*:

   NUM4

9. Numeric-keypad gray keys: NUM–, NUM+, and NUM*

10. Other named keys: BKSP, TAB, ESC, SPACEBAR, and ENTER
11. Combinations:
    a. ALT+ followed by items 1, 2, 4, 5, 9, or 10
    b. CTRL+ followed by items 2, 4, 7, 8, 9, or 10
    c. SHIFT+ followed by items 4, 7, 8, 9, or 10

Tandy 1000 If you have a Tandy® 1000, the following additional keystrokes are recognized:

1. Function keys: F11 and F12
2. CTRL+ followed by UP and DOWN
3. ALT+ followed by UP, DOWN, LEFT, and RIGHT
4. SHIFT+ followed by UP, DOWN, LEFT, and RIGHT

101-key enhanced keyboard If you have a 101-key enhanced keyboard, all the keystrokes recognized for the Tandy 1000 are supported, as well as the following: NUMENTER, NUM\, and all CTRL, ALT, and SHIFT combinations of these two keys. In addition, the enhanced keyboard has a duplicate set of item 7 above, which is not affected by the state of NUMLOCK. All CTRL, ALT, and SHIFT combinations of these keys are recognized.

Example

For example, the function *Savecur* is assigned to the keystroke CTRL+B in this manner:

1. Invoke the *Arg* function (press ALT+A)
2. Enter the function and keystroke as the *textarg* by typing the following:

   ```
   savecur:CTRL+W
   ```

3. Invoke the *Assign* function (press ALT+=)

   Note that function names are not case-sensitive, so you can enter them as all lowercase.

From this point on, pressing CTRL+W invokes the *Savecur* function and saves the current cursor position. You can make this assignment automatic by placing it in the TOOLS.INI file, as explained in Chapter 7.

6.2.2 Viewing and Changing Function Assignments

The <assign> pseudo file shows you what function assignments and switch values are in effect at any time during the editing session. This file lists all functions in alphabetical order along with the keys to which they are assigned. Use any of the following methods to get to this screen:

- Press SHIFT+F1 to get the initial Help screen. Then select Current Assignments.
- Use the *Setfile* function (F2), giving `<assign>` as a text argument.
- Give the *Assign* command (ALT+A), entering a question mark (?) as a text argument.

Once you get to the assignments screen, you can scroll through the information as you would through any file. Use the *Setfile* function (F2) to return to your original file.

You can also change assignments from within this file by following these steps:

1. Move the cursor to the line that contains the function assignment you wish to change.
2. Edit the line so that it contains a new function assignment.
3. Move the cursor to a different line. If the assignment you entered is syntactically correct, the editor highlights the line to show that the new assignment was accepted. If the assignment was incorrect, the editor restores the line to its previous state and reports an error message.

You can execute a save operation while in the <assign> pseudo file by giving the *Arg Arg Setfile* command. This command (which normally saves a file to disk under its own name) directs the editor to write any new assignments you have made to the TOOLS.INI file. Each time it starts, the editor automatically recognizes function assignments placed in this file. See Chapter 7, "Switches, Assignments, and the TOOLS.INI File," for more information about TOOLS.INI.

The *Tell* function (CTRL+T) gives you an alternative way of seeing what function is assigned to any given keystroke. Give the command *Arg Tell* (ALT+A CTRL+T), then enter a keystroke. The editor prints the name of the function assigned to this keystroke on the dialog line.

For more information on the *Tell* function, see Appendix A, "Reference Tables."

6.2.3 Disabling a Keystroke

To disable a keystroke so it invokes no function at all, assign the function *Unassigned* to the keystroke. As with other function assignments, use the *Arg textarg Assign* command. The argument *textarg* uses the following syntax:

unassigned:*key*

Here, *key* is the keystroke you want to remove.

For example, to disable the keystroke CTRL+A, perform the following steps:

1. Invoke the *Arg* function (press ALT+A)
2. Enter the function name as *Unassigned* and the keystroke by typing the following: `unassigned:CTRL+A`
3. Invoke the *Assign* function (press ALT+=)

After these steps are carried out, pressing CTRL+A does not invoke any functions. (Disabling a keystroke is temporary if you have set the key assignment in the TOOLS.INI file. For permanent results, change the TOOLS.INI file.)

6.2.4 Making a Keystroke Literal

Many text editors allow you to enter only alphanumeric characters and punctuation into a file. However, the Microsoft Editor makes it easy for you to insert special characters.

Each keystroke corresponds to an ASCII value. When a keystroke is considered a "literal" or "graphic," pressing the key causes the editor to place the corresponding value into the file. For example, if you make the keystroke CTRL+D a literal key, the editor places a character with the decimal value 4 into the file each time you press CTRL+D.

NOTE *When you use a special keystroke (such as F1 or ALT+A) as a literal, it loses any special meaning it would otherwise have and will not invoke a function. If the special value is one that can be printed (for example, if it is a printable extended-ASCII character), the editor displays the corresponding character on the screen.*

The Microsoft Editor provides two methods for converting a keystroke to a literal character. The first method is to use the *Quote* function. When you invoke

the *Quote* function (CTRL+P), the next key that you press—regardless of whatever special meaning it may otherwise have—is regarded as a literal character.

The second method is to assign the *Graphic* function to the keystroke. This uses the same syntax as any other function assignment: *Arg textarg Assign*. Assigning *Graphic* cancels any previous assignment to the keystroke, and causes the editor to consider the keystroke a literal character. For example, to insert a formfeed character in the file whenever CTRL+L is pressed, first follow these steps:

1. Invoke the *Arg* function (press ALT+A)
2. Enter the function *Graphic* and the keystroke as the *textarg* by typing the following: `graphic:CTRL+L`
3. Invoke the *Assign* function (press ALT+=)

The choice between assigning the *Graphic* function or invoking the *Quote* function depends on the situation. If you want to enter a special value into the file repeatedly, it is often easier to assign the *Graphic* function to the keystroke—since you only need to do it once. However, use of the *Quote* function provides you with more control.

By default, the *Graphic* function is assigned to all alphanumeric characters.

6.3 Creating Macros within the Editor

The fastest way to create a new editing function for the Microsoft Editor is to create a macro. This editing function can be as simple as inserting a long word or phrase, or it can involve complex operations.

There are two ways to create a macro. The easiest is to use the *Record* function to automatically record a series of actions. You can also enter a macro directly. The second method requires more knowledge of syntax, but allows you to use sophisticated features, such as conditionals.

After you have used the editor for awhile, you may want to use both methods. You can create a simple function by recording a macro, then increase the macro's power by editing it directly, using the techniques described in Sections 6.3.2–6.3.8.

Once a macro is defined and assigned to a keystroke, you can see how the macro is defined by using the *Tell* function. Give the *Arg textarg Tell* command (ALT+A *textarg* CTRL+T), in which *textarg* is the name of the macro. For more information on the *Tell* function, see Appendix A, "Reference Tables."

The maximum number of macros that can be defined at one time is 1,024.

6.3.1 Recording a Macro

Much like a tape recorder, the *Record* (ALT+R) function tells the editor, "make a record of all editing commands until I tell you to stop." After you stop the recording, the editor creates a macro function consisting of all the editing commands you just gave. When you invoke this macro, the editor plays back these commands in the order you gave them.

While the editor is recording commands, the letters `REC` appear at the end of the status line.

The *Record* function starts a recording and gives the macro the default name **recordvalue** when the recording is done. Invoking *Record* again turns the recording off.

The following steps are the quickest way to create a macro:

1. Invoke the *Record* function (ALT+R) to start the recording.
2. Execute the series of actions you wish to record.
3. Turn off the recording by invoking *Record* (ALT+R) again.
4. If **recordvalue** is not already assigned, assign it to a keystroke as described in Section 6.2, "Assigning Functions to Keystrokes."

After you complete these steps, the editor associates the new macro with the keystroke that **recordvalue** is assigned to. Whenever you press this key, the editor plays back the editing commands you gave in Step 2 above.

Other variations of the *Record* function are shown below:

Command (and Default Keystrokes)	**Description**
Arg textarg Record (ALT+A *textarg* ALT+R)	Turns on a recording. When the recording is finished, the macro is given the name specified in the text argument.
Meta Record (F9 ALT+R)	Turns on a recording in which commands you give are recorded but not executed. This is in effect a silent recording.
Arg Arg textarg Record (ALT+A ALT+A *textarg* ALT+R)	Turns on a recording, but if the specified macro already exists, the editor appends editing commands to the macro instead of replacing it.

By using the window and file commands described in Chapter 4, you can open a second window and load the <record> pseudo file. This file dynamically displays the value of the current recording.

By opening the <record> pseudo file in a separate window, you can watch the editor record the macro as you create it. The macro writes out the actual name of each editing command you give. As a result, the editor displays a macro definition using the syntax described in the next section.

6.3.2 Entering a Macro Directly

A macro is nothing more than a predefined series of functions and/or literal text. This fact makes the syntax of most macros almost self-evident: you build a macro by using the same syntax used throughout this manual. The most advanced macros, however, use the return value of functions to alter control flow. This special feature is presented in Section 6.3.8, "Macros That Use Conditionals."

To define a macro directly, follow these steps:

1. Choose a name. The name should not be a macro or function name already in use.
2. Enter the macro definition by using the *Arg textarg Assign* command (ALT+A *textarg* ALT+=), in which *textarg* has the following syntax:

   macroname:=*list*

   No spaces should separate *macroname* from the definition symbol (:=). The *list* contains function names and text strings enclosed in double quotes, as explained in the next section.
3. Assign the macro with the command *Arg textarg Assign*, in which the *textarg* has the following form:

   macroname:*keystroke*

The last step is optional. If you want to build a nested macro as described in Section 6.3.5, only the last macro defined needs to be assigned to a keystroke.

Instead of assigning a macro to a keystroke, you can execute the macro by giving its name as input to the *Execute* function, as explained in Section 6.3.4.

Example

The following sequence of actions defines a macro called `InsPhrase`, which inserts a certain sentence. The simplest macros insert a fixed string of text—these macros are useful because they can save you a lot of typing. This macro is then assigned to the key CTRL+F10. After you do the following action, the editor inserts the words "This is a sentence." at the current cursor position, every time you press CTRL+F10.

1. Invoke the *Arg* function (press ALT+A)
2. Enter the macro name and definition by typing the following:

```
InsPhrase:="This is a sentence."
```

3. Invoke the *Assign* function (press ALT+=)
4. Invoke the *Arg* function (press ALT+A)
5. Assign this macro to a keystroke by typing the following:

```
InsPhrase:CTRL+F10
```

6. Invoke the *Assign* function (press ALT+=)

6.3.3 Building the Macro List

Each item in a macro-definition list is either a function name or a string of text. The function names are the same as used throughout this manual but are not case sensitive. You can enter them all lowercase, all uppercase, or any combination.

Each string of text must be enclosed in quotes. Embedded quote marks are represented as \", and embedded backslashes are represented as \\.

Example

The following text arguments are all valid macro definitions:

```
Callfun:="x = QuadCalc(a, b, c);"
Callfun2:=linsert begline "x = QuadCalc(a, b, c);"
Movedown:=arg "15" plines
Putcomm:=begline "/*  " endline "  */"
Join:=endline right arg sdelete
Del3:=arg right right right ldelete
Join2:=savecur endline right arg sdelete restcur
endword:=arg arg "( !.!$!\\:!;!\\)!\\(!,)" psearch
```

The rest of this section examines each of these macro definitions.

```
Callfun:="x = QuadCalc(a, b, c);"
```

The example above defines a macro that inserts the text "x = QuadCalc(a, b, c);" whenever invoked. This macro is simple, yet useful. It replaces the typing of a phrase with a single editing function.

Executing this macro has the same effect as typing the phrase. The precise effect of this macro therefore depends on whether insert mode is on or off.

```
Callfun2:=linsert begline "x = QuadCalc(a, b, c);"
```

The example above uses both function names and text to define a new editing function. The effect of invoking this macro is precisely the same as the effect of invoking *Linsert*, invoking *Begline*, and then typing the text inside the quotes. The macro inserts a new line and places the text at the beginning of this line.

```
Movedown:=arg "15" plines
```

Like the example before it, the example above uses a combination of functions and a string of text. However, because the *Arg* function precedes the text, the text does not appear on screen. Instead, the text becomes an argument to the *Plines* function. This macro moves the window down 15 lines.

```
Putcomm:=begline "/*  " endline "  */"
```

The example above uses a combination of functions and text to insert C-language comment marks at the beginning and end of a line.

```
Join:=endline right arg sdelete
```

The example above defines a macro that joins the current line to the next line. Recall that the sequence *Arg Sdelete* deletes all text to the right of the cursor, then joins lines. This macro does not delete any text. First, it moves the cursor to the end of the line. Next, it moves one space to the right to leave a space before the next line. It then executes *Arg Sdelete*.

You can use the cursor-movement functions (*Up*, *Down*, *Right*, *Left*) within macros. The next macro uses these functions to build an on-screen argument.

```
Del3:=arg right right right ldelete
```

The example above deletes three characters. The middle three arguments create a cursor-movement argument that is passed to the *Ldelete* function.

```
Join2:=savecur endline right arg sdelete restcur
```

The example above performs almost the same activity as the `Join` macro presented earlier. However, this macro begins by saving the cursor position with *Savecur*. After the lines are joined, the macro restores the original cursor position by executing *Restcur*. This macro works even if *Savecur* and *Restcur* have not been assigned to keystrokes.

```
endword:=arg arg "( !.!S!\\:!;!\\)!\\(!,)" psearch
```

This last example uses the regular-expression syntax described in Chapter 5. The *Arg Arg* syntax directs the *Psearch* function to treat the text argument as a regular-expression pattern. This pattern, in turn, causes the editor to find the next space, period (.), end of line ($), colon (:), semicolon (;), right parenthesis ( )),

left parenthesis (( ), or comma (,). Some characters must be preceded by a backslash (\ ) in order to be interpreted literally. Furthermore, when you give a text argument inside a macro list, you must use two backslashes in a row (\\ ) to indicate a single backslash.

The effect of this macro is to go to the end of the current word. By rewriting the regular expression, you control what constitutes the end of a word. This macro can be especially useful when nested inside other macros, as described in Section 6.3.5.

6.3.4 Executing a Macro List Directly

Instead of defining a macro and then assigning it to a keystroke, you can execute a macro-definition list directly, by using the *Execute* function. This function, which by default is assigned to F7, takes a function name, macro name, or macro-definition command list as a text argument.

For example, the following sequence finds the next occurrence of the word `fluke`:

1. Invoke *Arg* (press ALT+A)
2. Type the following string: `arg "fluke" psearch`
3. Invoke *Execute* (press F7)

You can also give the name of a macro as input to the *Execute* command.

6.3.5 Building Macros from Other Macros

Macros can contain references to previously defined macros. Since a macro definition must be contained on one line (except in the TOOLS.INI file, where you can use a line-continuation character), you may need to break up a macro definition into several smaller macros as shown in the example below. Only the final macro definition need be assigned to a keystroke. Each of the following lines is entered one at a time using the *Arg textarg Assign* command:

```
head1:=arg "3" linsert "/************************"
head2:=newline "** Routine:"
head3:=newline "************************/"
header:=head1 head2 head3
header:alt+h
```

The example above inserts three blank lines (by passing the argument 3 to the *Linsert* function), and then inserts the given strings of text. The macro is then

assigned to ALT+H. This macro is an example of one that automates the creation of comment blocks for C programs.

NOTE *Nested macros are cumbersome to enter while running the editor. Usually, you'll want to enter them once in your TOOLS.INI file so that they are available automatically, as explained in Chapter 7.*

6.3.6 Handling Prompts within Macros

Some commands prompt the user for confirmation. For example, the *Meta Exit* command (exit without saving) asks the user if he really wants to exit. These questions take the answer yes (Y) or no (N).

By default, macros assume an answer of yes. For example, if you assigned *Meta Exit* to a macro and then executed the macro, the editor would not prompt for confirmation. Instead, the editor would assume an answer of yes and proceed to exit without saving.

You can control the answers to prompts within macros by using the following operators, each of which applies to the immediately preceding function:

Operator	Description
<	Asks the user for confirmation. (If not followed by another < character, prompts user for all further questions.)
<Y	Assumes an answer of yes.
<N	Assumes an answer of no.

You can use a series of operators. For example, consider the following macro:

```
newfile:=arg "newfile.txt" setfile <y <n
```

The macro `newfile` assumes an answer of yes for the first prompt, and no for the second. If `newfile` requires only one prompt, it ignores the second operator, `<n`.

If the last prompt character that appears is <, the editor prompts the user directly for all remaining yes/no questions.

You can use the prompt characters throughout your macro. For example:

```
newfile:=refresh < arg "newfile.txt" setfile <y <n <
```

6.3.7 Macros That Take Arguments

Macros have no explicit syntax for accepting user-defined arguments. However, if you enter an argument and then invoke a macro, the argument is passed to the first function that accepts an argument:

```
tripleit:=copy paste paste
```

In the example above, you can invoke *Arg* and use cursor movement to highlight an argument. Then invoke `tripleit`. Your highlighted argument is passed to the *Copy* function, which copies the argument to the Clipboard. The macro then executes the *Paste* function two times. As a result, two additional copies of the argument are added to the file.

If you do not highlight an argument before invoking `tripleit`, the macro responds by printing two more copies of the current line. By default, `tripleit` works on the current line because *Copy* selects the current line when no argument is given.

When you execute the *Copy* or *Delete* function at the beginning of a macro, the highlighted argument is stored in the Clipboard. You can then load the <clipboard> pseudo file and manipulate the argument itself, as in the following macro:

```
addto1:=copy arg "<clipboard>" setfile
addto2:=arg arg "TMP.TXT" setfile
addto3:=arg "type TMP.TXT >> BIG.TXT" shell
addto4:=arg "del TMP.TXT" shell setfile
addto:=addto1 addto2 addto3 addto4
```

The `addto` macro appends the highlighted argument to the file BIG.TXT. If you do not give an argument, the macro appends the current line. The `addto` macro works by executing `addto1`, `addto2`, `addto3`, and `addto4`, each of which does some of the work of adding to the highlighted area. The following list describes the steps of the `addto` macro:

1. Copy argument to the Clipboard, then load the <clipboard> pseudo file.
2. Save the contents of the current file (which is now <clipboard>) to the file TMP.TXT.
3. Execute a DOS shell that appends TMP.TXT to BIG.TXT.
4. Execute a DOS shell that deletes TMP.TXT, then use *Setfile* to return to the original file.

Another way to respond to an argument is to put the *Replace* function at the beginning of your macro. Replacements then take place throughout the

highlighted area. For example, the following macro replaces each occurrence of the period (.) with a blank space:

```
blankout:=replace "." newline " " newline
```

The *Newline* function is necessary because the *Replace* function prompts for search-and-replace strings.

To use the macro above in a practical way, invoke *Arg* twice to enable regular expressions, highlight an area of text, and then execute `blankout`. Assuming that the editor is using Unix regular-expression syntax, the effect is that each character in the highlighted area is replaced by a blank space. The effect is different from *Delete* because the space occupied by the text is not removed, simply replaced with blanks.

WARNING *Using Arg twice and then executing* `blankout` *without a highlighted region replaces the rest of your file with blank spaces. Make sure you use a highlighted argument.*

6.3.8 Macros That Use Conditionals

You can write macros that execute different actions depending on certain conditions. These macros take advantage of function return values. A "return value" is simply a piece of information that a function passes back after it is executed. Editing functions always return the value TRUE (nonzero) or FALSE (zero).

Each editing function has different criteria for determining what to return, but usually an editing function returns TRUE if the function is successful, or FALSE if it fails. For example, a cursor-movement function fails if the cursor does not move.

Table A.4, in Appendix A, gives a complete list of function return values. You use these return values with the syntax described in Table 6.1.

Table 6.1 Macro Conditionals

Conditional	Description
:>*label*	Defines a label that can be referenced by other macro conditionals.
=>*label*	Causes a direct transfer to *label*. If *label* is omitted, then the current macro exits.
->*label*	Causes a direct transfer to *label* if the previous function returns FALSE. If *label* is omitted, then the current macro exits on FALSE.
+>*label*	Causes a direct transfer to *label* if the previous function returns TRUE. If *label* is omitted, then the current macro exits on TRUE.

The table above refers to a macro exiting. In this context, exiting a macro only affects the current macro. If macro `doall` executes macro `submacro`, when `submacro` exits, control returns to `doall`, which continues execution.

Examples

The first example turns insert mode on. The only function that turns insert mode on or off is the *Insertmode* function. This function toggles between the on and off condition, and returns the new state of insert mode:

Initial state	Effect of Insertmode
Off	Turn insert mode on; return TRUE.
On	Turn insert mode off; return FALSE.

Unfortunately, testing the state of insert mode may turn insert mode off, even though the goal is to leave it on. Fortunately, a conditional macro can turn insert mode back on again as appropriate:

```
turnon:=insertmode +> insertmode
```

This macro consists of three simple steps:

1. `insertmode` toggles the state of insert mode. This function returns TRUE if insert mode is now on.
2. The `+>` operator exits the macro if the last function executed returned TRUE. In this case, a return value of TRUE indicates that insert mode is now on, so the macro exits on TRUE.
3. If the last function returned FALSE, the macro continues to the last step, which executes `insertmode` again. If the macro reaches this step, insert mode was initially on but the first step turned it off. This step turns insert mode back on again.

The macro defined above, `turnon`, is highly useful when used within nested macros. Define `turnon` first. Then place `turnon` at the beginning of a macro definition whenever the macro needs to assume that insert mode is on.

NOTE *In the next chapter, you'll learn how to use the* ***enterinsmode*** *switch to achieve the same result as the macro defined above..*

The next example turns insert mode on temporarily, executes the predefined macro `cmd`, and then leaves the editor in the same state that it was in before the macro was executed.

```
icmd:=insertmode +>on insertmode cmd => :>on cmd insertmode
```

The macro executes the commands in the following order:

1. `insertmode` toggles the state of insert mode, so if insert mode was off, it is now on, and vice versa. This function returns TRUE if insert mode is on after the function is executed, and FALSE otherwise.
2. `+>on` transfers control to the label `on` if the previous function returned TRUE—in other words, if insert mode is now on.
3. `insertmode` turns insert on. If the macro reaches this point, insert mode was originally on but the first step turned it off and returned FALSE. This step turns insert mode back on.
4. `cmd` executes the predefined macro.
5. `=>` exits the macro. Insert mode was on when the macro started, it is now back on, and `cmd` has been executed. Therefore the macro is done.
6. `:>on` defines a label. Execution transfers here directly from the item `+>on` if insert mode was turned on.
7. `cmd` executes the predefined macro.
8. `insertmode` turns insert mode back off. Since execution flowed through to this point, insert mode must have been off when the macro began—insert mode is turned back off now.

CHAPTER 7

Switches, Assignments, and the TOOLS.INI File

7

A "switch" is a variable that controls some condition of the editor. For example, changing the **fgcolor** switch alters the foreground color of the editor, and changing the **height** switch alters the number of lines on the screen. Although switches come in three varieties (numeric, text, and Boolean), all three kinds of switches can be controlled with the *Assign* function.

Usually, the most convenient way to work with switches is to set them in the initialization file, named TOOLS.INI. This file contains switch values, function assignments, and macros for the Microsoft Editor to automatically assume at the beginning of each session. This file can be used to initialize settings for other products. Therefore, all the settings for the editor have to be put under a special heading called a "tag."

If you are interested in using TOOLS.INI right away to make the editor automatically start up with your own function assignments and macros, turn to Section 7.4, "Sample TOOLS.INI File."

This chapter covers the following topics:

- Syntax for switch settings
- Using switches to configure the editor
- Special syntax for text switches
- Sample TOOLS.INI file
- The structure of the TOOLS.INI file
- Configuring on-line Help

7.1 Syntax for Switch Settings

To change the behavior of the editor, set a switch with the *Assign* function or in the TOOLS.INI file. You set a switch differently depending on the type of the switch. Each switch is one of three types: numeric, text, or Boolean.

To set a numeric or text switch, give the assignment

switch:*value*

in which *switch* is the name of the switch and *value* is either a string of digits (in the case of a numeric switch), or any string of text (in the case of a text switch). Most numeric switches use decimal digits. However, the color switches use hexadecimal digits, as described in Section 7.2.3.

Boolean switches can be off or on. To turn a Boolean switch on, give either of the following assignments:

switch:
switch:**yes**

To turn a Boolean switch off, prefix the letters **no** to the assignment or type **no** after the colon:

no*switch*:
switch:**no**

A switch assignment looks similar to a function assignment. However, a switch assignment has a different effect. A function assignment changes the meaning of a keystroke, whereas a switch assignment alters a basic condition, such as screen color.

Examples

The following examples show a numeric, text, and four Boolean assignments, respectively:

```
tabstops:8
backup:none
case:
nocase:
case:yes
case:no
```

You could make any of these assignments by placing them in the TOOLS.INI file, or by using the *Arg textarg Assign* command. For example, to set **tabstops** to 8, follow these steps:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following switch assignment: `tabstops:8`
3. Invoke the *Assign* function (press ALT+=)

NOTE *Some text switches recognize special characters. These characters are never required, but can be convenient. See Section 7.3, "Special Syntax for Text Switches," for more information.*

7.2 Using Switches to Configure the Editor

This section explains how to use some of the more common switches. You can follow the instructions to get a feel for how to adjust conditions within the editor. For example, you may want to change screen colors, screen height, or tab behavior. Sections 7.4 and 7.5 explain how to make these settings automatic by placing them in the TOOLS.INI file.

Sections 7.2.1–7.2.8 cover the following topics:

- Changing start-up conditions
- Changing scrolling behavior
- Setting screen colors with **fgcolor**
- Setting colors for other parts of the screen
- Changing the look and feel of Help
- Controlling use of tabs
- Changing how the editor handles trailing spaces
- Changing screen height

7.2.1 Changing Start-Up Conditions

By default, the editor starts with insert mode off and status line displaying window position (rather than cursor position). To start with insert mode on, place the following statement in the [M] section of your TOOLS.INI file:

```
enterinsmode:
```

You can also set this switch by executing these steps:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following switch assignment: `enterinsmode:`
3. Invoke the *Assign* function (press ALT+=)

Setting this switch while in an editing session turns insert mode on regardless of the current state of insert mode.

The status line at the bottom of the screen displays window position by default. The window position gives the file coordinates (by line and column) of the top left corner. To change this field so that it displays cursor position instead, follow these steps:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following switch assignment: `displaycursor:`
3. Invoke the *Assign* function (press ALT+=)

The editor numbers rows and columns beginning with the number one. Cursor position is (1,1) when the cursor is at the very beginning of the file.

The **enterinsmode** and **displaycursor** switches are both examples of Boolean switches. To turn these switches off, follow the directions above but precede the switch with the word **no**.

7.2.2 Changing Scrolling Behavior

Two switches control scrolling behavior in the Microsoft Editor: **vscroll** and **hscroll**. Each time you attempt to move the cursor off the top or bottom of the screen, the editor moves the editing window up or down a certain distance. This distance is set by the value of **vscroll**.

For smooth vertical scrolling, set **vscroll** to 1:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following switch assignment: `vscroll:1`
3. Invoke the *Assign* function (press ALT+=)

For smooth horizontal scrolling, set **hscroll** to 1. Give the *Arg textarg Assign* command (ALT+A *textarg* ALT+=), in which *textarg* is `hscroll:1`.

7.2.3 Setting Screen Colors with fgcolor

The **fgcolor** switch controls the most important screen colors: background and foreground colors of editing windows. The background color is the primary color of a region of the screen. The foreground color is the color of characters that appear in this region.

The **fgcolor** switch uses one numeric value to control both background and foreground colors. The other color switches (**errcolor**, **hgcolor**, **infcolor**, and **stacolor**) all work in the same way: each sets a background and foreground color according to a single numeric value.

The editor reads a color switch as a two-digit hexadecimal number. The first digit sets the background color; the second sets the foreground color. Table 7.1 states the digits that correspond to each color.

In the case of background colors, the digits 8–F correspond to the digits 0–7, except that they make the foreground text blink on and off.

Table 7.1 Colors and Numeric Values

Color	Value
Black	0
Blue	1
Green	2
Cyan	3
Red	4
Magenta	5
Brown	6
Light Gray	7
Dark Gray	8
Light Blue	9
Light Green	A
Light Cyan	B
Light Red	C
Light Magenta	D
Light Yellow	E
White	F

Each two-digit number describes some combination of background and foreground colors. The list below gives a few examples:

Number	Meaning
17	Blue (1) background, light gray (7) foreground
07	Black (0) background, light gray (7) foreground
2F	Green (2) background, white (F) foreground
24	Green (2) background, red (4) foreground

NOTE *Only color adapter cards support all the colors listed above. If you have a monochrome adapter or monochrome monitor, the only colors available are white (F), black (0), and light gray (7). All other colors are treated as white.*

Example

At any time during the editing session, you can set new screen colors (although to set these colors permanently, you should use TOOLS.INI). To set a blue background with a light gray foreground, execute the following steps:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following switch assignment: `fgcolor:17`
3. Invoke the *Assign* function (press ALT+=)

7.2.4 Setting Colors for Other Parts of the Screen

All color switches interpret digits in the same way. However, each color switch controls a different region of the screen, as described in the list below:

Switch	Description
errcolor	Controls the colors used to display error messages on the dialog line.
hgcolor	Controls the colors in a region of text highlighted by a search command.

infcolor	Controls the colors used for informative text: messages (other than error messages) that the editor displays on the dialog line, as well as strings to be replaced that are located by *Qreplace*. The editor displays messages after executing an operation, such as saving a file or loading a new file. These colors provide the default background for the dialog line, and also control the color of "insert" and "meta" when they appear on the status line.
selcolor	Controls the color of text that the user highlights as a cursor-movement argument.
stacolor	Controls the colors of most items on the status line.
wdcolor	Controls the color used to draw window borders.

The following procedure sets a black (0) background for error messages, combined with a red (4) foreground. Error messages will appear in red letters against a black background:

1. Invoke the *Arg* function (press ALT+A)
2. Type the following switch assignment: `errcolor:04`
3. Invoke the *Assign* function (press ALT+=)

7.2.5 Changing the Look and Feel of Help

The information in this section applies only to systems configured to use the powerful Microsoft Help engine. If Help is not installed when you press SHIFT+F1, a message appears saying that you have not installed Help yet. Run the installation program for your Microsoft language, or follow the directions in Section 7.6, "Configuring On-Line Help."

By default, Help splits the screen and displays Help information in a separate window. This behavior has the advantage of letting you view both the current file and Help information at the same time. You can move the cursor between windows by pressing F6. You can also copy examples from the Help window and paste them into an editing window.

You can change the behavior of Help so that it does not split the screen. Instead, Help will save the current file (if the **autosave** switch is on) and load Help information into the current editing window. This behavior has the advantage of using more room to display Help information.

The split-screen behavior is controlled by the **helpwindow** switch. This switch is a Boolean switch; it can be either on or off. Turning the switch off prevents the editor from splitting the screen to display Help information:

1. Invoke *Arg* (press ALT+A)
2. Type the following switch assignment: `nohelpwindow:`
3. Invoke the *Assign* function (press ALT+=)

Conversely, to turn the split-screen behavior back on, give the *Arg textarg Assign* command, in which *textarg* is `helpwindow:`.

NOTE *When **helpwindow** is on, use the Cancel function (ESC) to close the Help window. When **helpwindow** is off, use the Cancel function (ESC) to return to the previous file.*

You can also control the colors used in the Help window. The writer of a help file can designate text as foreground, bold, italicized, underlined, or as warning text. You cannot change these designations. However, you can easily change the choice of colors used to illustrate each kind of text:

Switch	Meaning
fgcolor	Foreground color, which controls the color of normal text. This switch also controls the foreground color of editing windows.
helpboldcolor	Color of text designated as bold.
helpitalcolor	Color of text designated as italicized.
helpundcolor	Color of text designated as underlined.
helpwarncolor	Color of text used for a "warning" note. Also controls color of highlighted cross-references.

All of the switches above are set the same way the **fgcolor** switch is, as explained in Section 7.2.3.

7.2.6 Controlling Use of Tabs

The Microsoft Editor provides two basic ways of working with tab characters: you can either treat tabs as real characters, or you can have the editor convert each tab into a series of spaces. This behavior is controlled by the **realtabs** switch.

The **realtabs** switch is a Boolean switch that by default is on, causing the editor to treat each tab (ASCII 9) as an independent character. Unlike other characters, a tab may correspond to several positions on the screen. If you place the cursor on a tab character and press RIGHT, the cursor jumps to the next tab column. Tab columns occur at regular intervals, as determined by the **filetab** switch.

As you add and delete characters, the editor maintains the tab-column alignment of the text.

Whether or not **realtabs** is on, the *Tab* function is simply a movement function, and it does not insert tabs into a file. To use the TAB key to directly insert tabs, put the following assignment in your TOOLS.INI file, or give it as input to the *Assign* command:

```
graphic:tab
```

This assignment makes the TAB key a graphic character. Each time you press TAB, you directly place an actual tab character into the file.

If you turn the **realtabs** switch off, the Microsoft Editor translates tab characters into spaces. This behavior only affects the individual lines that you modify. If you read a file but make no changes, the tab-to-space conversion does not alter the file on disk, even if you execute a save operation.

The editor may also convert spaces to tabs, according to the setting of the **entab** switch, described below. This behavior also only affects the individual lines that you modify.

The **tabdisp** switch is useful for viewing the effect of writing to a disk file. If you set **tabdisp** to a number other than 0 or 32, the editor shows you which spaces will be compressed into a tab character at the next write to the disk file. All such spaces are displayed as the ASCII equivalent of the value of **tabdisp**.

If **realtabs** is on, each position in a tab field displays this same character.

For example, the following steps set **tabdisp** to 249:

1. Invoke *Arg* (press ALT+A)
2. Type the following switch assignment: `tabdisp:249`
3. Invoke the *Assign* function (press ALT+=)

After you carry out these steps, the editor displays the ASCII equivalent of 249 in place of each position in a tab field.

The following list describes the meaning of the four tab-handling switches:

Switch	Description
realtabs	Controls whether or not tabs are treated as real characters, as described above. If on, tab columns are aligned according to the **filetab** switch. If **realtabs** is off, every tab read is treated as a series of spaces, according to the **filetab** switch.
entab	Controls the extent to which the editor converts a series of tabs and spaces to tabs when saving a file. Only the lines you modify during the editing session are affected by space-to-tab conversion. A value of 0 means the editor does not replace spaces by tabs. A value of 1 (the default) means the editor can replace a series of tabs and spaces by tabs when the spaces fall outside of quoted strings. A value of 2 means all series of tabs and spaces can be replaced by tabs. The **entab** switch determines what kind of space-to-tab replacements are possible. Whether a replacement is made in any given case depends on the position of the spaces, and on the **filetab** switch, described next.
filetab	Controls the physical (disk-file) meaning of tab characters. If **realtabs** is on, the **filetab** switch determines tab alignment. If **realtabs** is off, the **filetab** switch determines how the editor translates tab characters to spaces when a line of text is modified. If **entab** is set to 1 or 2, **filetab** also determines how the editor translates spaces to tabs when you save the file to disk. (Only modified lines are affected.)

The value of the switch gives the number of spaces associated with each tab column. For example, the setting "filetab:4" assumes a tab column every 4 positions on each line. Every time the editor finds a tab character (ASCII 9), it loads the buffer with the number of spaces necessary to get to the next tab column. The default value of **filetab** is 8.

tabalign

If **realtabs** and **tabalign** are both set, the cursor automatically moves to the first column position of a tab character when it is placed anywhere within a tab character. Cursor movement corresponds to the actual characters in the file.

When not set (the default), the cursor may be placed anywhere in any column of a tab character. If a character is typed at this position, sufficient leading blanks will be inserted to assure that the character actually appears in this column position. Cursor movement is independent of the actual characters in the file.

tabstops

Determines the size of columns associated with the *Tab* and *Backtab* cursor-movement functions. It has no affect on actual tab characters. The default value of **tabstops** is 4.

7.2.7 Changing How the Editor Handles Trailing Spaces

A "trailing space" is a space character located to the right of the last printable character on a line. A trailing space is normally invisible. By default, the editor deletes all trailing spaces on each line you modify.

However, you can change this behavior by resetting the value of the **trailspace** switch. This switch is Boolean switch; it can be either on or off. To turn this switch on, give the *Arg textarg Assign* command (ALT+A *textarg* ALT+=), in which `trailspace:` is the *textarg*. When you wish to turn this switch off, give the *Arg textarg Assign* command, in which `notrailspace:` is the *textarg*.

When this switch is on, any space you type at the end of a line remains in the file as a trailing space. You can verify the existence of trailing spaces by invoking the *Endline* function (press END). If the cursor moves past the last printable character when you invoke *Endline*, the line contains trailing spaces.

To display trailing spaces, set the **traildisp** switch to a number greater than 0. The editor displays the ASCII-character equivalent of the number you choose, in place of each trailing space. For example, invoke the *Arg textarg Assign* command in which *textarg* is `traildisp:1`.

The example above displays the ASCII equivalent of the number 1 in place of each trailing space. To restore normal display, set **traildisp** back to 0.

7.2.8 Changing Screen Height

Change the screen height by setting the **height** switch. This switch takes only a few values that are limited by the kind of graphics adapter card you are using:

Graphics Card	Legal Values
CGA or monochrome	23 (25-line mode)
EGA	23 (25-line mode) 41 (43-line mode)
VGA	23 (25-line mode) 41 (43-line mode) 48 (50-line mode)

Note that you assign to **height** a number two less than the desired mode. That is because **height** refers to the height of the editing window, not the full screen. (Two lines at the bottom of the screen are reserved for dialog and status.)

7.3 Special Syntax for Text Switches

The following text switches recognize special characters for referring to file names or directories:

- **extmake**, which can be repeatedly assigned to different compilation commands
- **readonly**, which gives a system-level command, executed whenever you attempt to overwrite a read-only file
- **load**, which loads a C extension as described in Chapter 8
- **markfile**, which loads file markers from a specially prepared file
- **helpfiles**, which specifies which .HLP files should be used by on-line Help

This syntax presented in the next two sections is never required, but is provided for convenience.

7.3.1 Special Syntax for extmake and readonly

The text switches **extmake** and **readonly** each interpret the following characters as the name of the current file:

%s

You can also use a more comprehensive syntax that lets you specify drive, file name, base name, or file extension. The syntax consists of the characters

%|*letters***F**

where *letters* consists of any of the following: p for path, d for drive, f for file base name, or e for file extension. For example, if you are editing the file c:\dir1\sample.c, and you make the switch assignment

```
extmake:c cl /Fod:%|pfF %|dfeF
```

then each time you give the command *Arg Compilation*, the editor performs the following system-level command:

```
cl /Fod:\dir1\sample c:sample.c
```

The expression `%s` is equivalent to `%|feF` except that the former is only accepted once in each assignment, whereas the latter can appear any number of times in the **extmake** switch assignment. The expression `%|F` is equivalent to `%|dpfeF`.

7.3.2 Special Syntax for load, markfile, and helpfiles

The **load**, **markfile**, and **helpfiles** switches each search for a file. These switches can also take the syntax

$*environ***:**

in which *environ* is the name of an environment variable recognized by the operating system. The editor will search directories listed in the environment variable to find the file. Environment variables are created with the system-level SET command. See your operating-system documentation for more information on the SET command.

For example, the following assignment causes the editor to search directories in the INIT environment variable for the file MARKERS.DAT:

```
markfile:$INIT:markers.dat
```

Note that the environment variable should be entered in uppercase letters.

7.4 Sample TOOLS.INI File

Each time you start the editor, it checks the [M] section of the TOOLS.INI file for any function assignments, switch assignments, and macro definitions you have placed there. (However, the editor may check a different section if the editor is not named M.EXE.)

You can change these settings at any time during the editing session by using the *Assign* command. However, using TOOLS.INI is more convenient for settings you want to use every time. Use TOOLS.INI to make the editor automatically start up with all the screen colors, tab behavior, function assignments, and special editing functions you want to make available for all editing sessions.

The editor loads settings from TOOLS.INI when it starts and when you invoke the *Initialize* function (SHIFT+F8) during the editing session. By giving a text argument, you can also use *Initialize* to load specific sections within TOOLS.INI.

NOTE *The editor checks the directories listed in the INIT environment variable for the location of the TOOLS.INI file. For example, if the TOOLS.INI file is in the directory C:\BIN, place the following statement in your AUTOEXEC.BAT file:* `SET INIT=C:\BIN.`

The example below shows a sample TOOLS.INI file. The next section explains each section of this file in depth. Note that text following the semicolon (;) is a comment.

```
[M MEP]                          ; Settings for both M and MEP

        enterinsmode:            ; Start up in insert mode
        backup:none              ; Perform no backup
        displaycursor:           ; Display cursor on status line
        fgcolor:17               ; Set primary and error colors
        errcolor:04

        join:=savecur endline right arg sdelete savecur   ; Macro

        savecur:ctrl+f5          ; Assignments to keys
        restcur:ctrl+f6
        join:alt+j

[M-C MEP-C]                      ; C specific section - enable with Initialize

        tabstops:3               ; Tab columns 3 wide within editor
        case:                    ; Case-sensitive searches

[NMAKE]

#  This is a comment line recognized by NMAKE
```

7.5 The Structure of the TOOLS.INI File

To create a TOOLS.INI file for the editor, you need to follow a few simple rules:

1. You must precede your editor settings by a tag, or they will not be recognized. A "tag" is a heading within the TOOLS.INI file that divides the file into sections.

   For example, you can use tags to create a section for the editor ([M] or [MEP], unless you have renamed the editor) and a section for another utility such as NMAKE, which looks for a section headed by the [NMAKE] tag. Each utility has its own syntax for interpreting TOOLS.INI statements, but all utilities must recognize tags. See the *Microsoft CodeView and Utilities User's Guide* for more information on NMAKE.

2. A semicolon (;) indicates that all text from the semicolon to the end of the line is a comment, and is ignored by the editor. To mark the beginning of a comment, a semicolon must either appear at the beginning of a line or be preceded by a space.

3. All macro definitions, function assignments, and switch assignments use exactly the same syntax used with the *Assign* command.

   However, you do not invoke *Arg* or *Assign*; simply enter the assignment itself. (See the sample TOOLS.INI file for clarification.)

4. You can use the backslash (\ ) as a line-continuation character to continue statements too long for one line. The backslash must be preceded by a space and can be followed only by trailing spaces and an optional comment. Normally, you should precede the backslash by two spaces.

The next few sections explain each part of the file—tags, comments, and assignments.

7.5.1 Creating Sections with Tags

Tags divide the TOOLS.INI file into sections. All statements are associated only with the tag they immediately follow. This feature allows programs other than the Microsoft Editor to use this file for configuration information. The most common way to use a tag is to simply include all assignments and macro definitions after an M or MEP tag (depending on the name of the editor):

[M]

When you run the Microsoft Editor under DOS 2. *x*, the editor always responds to the tag [M]. Otherwise, the tag should use the base file name of the editor. If the editor is named MEP.EXE, use the tag [MEP]. Use [M MEP] if you have both M.EXE and MEP.EXE.

IMPORTANT *The name you use for the main editor tag should appear in all other editor tags as well. For example, if the main editor tag is [EDIT] (because you renamed the editor to EDIT.EXE), all editor tags should begin with the word EDIT. Instead of using [M-3.20], you would use [EDIT-3.20].*

When the Microsoft Editor is started, the tagged sections are loaded in the following order:

1. Information used for all editing sessions.

   All of the statements in the [M] section are loaded. Remember to use a different tag if you rename the editor.

2. Information specific to the operating system.

   Depending upon the operating system you are running, one of the following tagged sections is loaded (if present):

   - [M-3.20] (MS-DOS)
   - [M-10.0] (OS/2 protected mode)
   - [M-10.0R] (OS/2 real mode)

   With the DOS or OS/2 version tag, you should insert the version number you are using. (OS/2 1.10 uses the tag 10.10.) You can combine operating-system tags by grouping them together within the brackets:

```
[M-3.20 M-3.30]                    ; For DOS versions
[MEP-10.0 MEP-10.10]               ; For OS/2 versions
[M-3.20 M-3.30 M-10.0R M-10.10R]   ; For real mode
```

3. Information specific to the display.

   Depending on the video display you are using, one of the following tagged sections is loaded (if present):

   - [M-mono]
   - [M-cga]
   - [M-ega]
   - [M-vga]

   You can also put statements for setting the screen dimensions and colors in these tagged sections.

   All tags can be combined so one section of statements applies to more than one tag:

```
[M-mono M-cga MEP-mono MEP-cga]
```

   The example above begins a section that is loaded if your computer has either a monochrome adapter or a CGA adapter.

Creating tags for a file extension

You can also create a tag with statements specific to a file extension. The form is

[M-*.ext*]

where *.ext* is an extension of up to three characters. Whenever you load a new file, or switch to editing a different file, the editor automatically searches TOOLS.INI for a tag that matches the new file extension. If it finds a matching tag, the statements following that tag are executed. If not, the existing configuration is retained.

This feature can be used to create operating environments that are specific to a particular type of text file or programming language. For example, the tag `[M-.FOR]` could precede a set of statements for editing .FOR (FORTRAN) files: a blue background, the right margin at column 72, and tabs set every three columns. C-language files with the .C extension could have their own set of statements following a `[M-.C]` tag: a magenta background, the right margin at column 128, and tabs set every five columns.

Creating tags for manually loaded statements

You can also use a tag to create a special section that is loaded manually with the *Arg textarg Initialize* command. These tags use the following syntax:

[M-*textarg*]

This feature lets you make special key assignments, load rarely used macros, or switch the operating environment to a special configuration, only when required.

7.5.2 Using Comments

The Microsoft Editor considers text from a semicolon (;) to the end of a line to be a comment. Comments are for documentation purposes only and are ignored by the editor.

To be considered part of a comment, the semicolon must either appear at the beginning of a line or else be preceded by a space. Thus, in the following line, the first semicolon is considered part of a command, and the second begins a comment:

```
extmake:asm masm /Zi /MX %s; ; "Compile" setting for MASM
```

Semicolons inside a quoted string do not begin a comment.

7.5.3 Line Continuation

Use the backslash (\ ) to continue a long macro definition to the next line. To be interpreted as the line-continuation character, the backslash must be preceded by a space. It should be the last character on the line except for spaces and comments. The editor does not interpret the backslash as the line-continuation character in the following statement:

```
qreplace:ctrl+\
```

The following statement illustrates valid use of line continuation:

```
findswitch:=psearch ->skip arg mark :>skip arg "<assign>"  \
            setfile begfile psearch
```

Note that the backslash should be followed by at least two spaces, unless you want the end of one line to be concatenated without a break to the beginning of the next. In the example above, two spaces are necessary to prevent the editor from viewing `"<assign>"setfile` as one item.

You can use line continuation to extend a single assignment statement over several lines.

7.5.4 Assignments and Macros

Function assignments, switch assignments, and macro definitions all use the same syntax within the TOOLS.INI file as they do with the *Arg textarg Assign* command. The only difference is you do not need to invoke the *Arg* or *Assign* functions.

The TOOLS.INI file is convenient for defining complicated macros, especially nested macros. By placing macro definitions in the TOOLS.INI file, you can view how you wrote a macro. Furthermore, you can easily modify a macro that doesn't work correctly by loading the TOOLS.INI file, making changes to your macro definitions, and reinitializing.

If you customized your copy of the Microsoft Editor when you installed it, a special TOOLS.INI was created that contains the special key assignments. This TOOLS.INI file also contains a number of useful macros from the TOOLS.PRE file. You may want to study TOOLS.PRE to see how macros are entered and which macros might be useful to you.

Use the *Initialize* function (SHIFT+F8) to reload your TOOLS.INI settings. When you give the *Initialize* function a *textarg*, it attempts to use settings from the following section:

[M-*textarg*]

However, when you invoke *Initialize* with no argument, the editor reloads settings in the TOOLS.INI file in the same way that it does on start-up.

You can create a macro to be executed whenever the editor starts up. Simply define a macro and assign it the name `autostart`. For example, when you place the following macro in the [M] section of the TOOLS.INI file, the editor transfers to the Clipboard and returns, thus making the Clipboard the previous file:

```
autostart:=arg "<clipboard>" setfile setfile
```

7.6 Configuring On-Line Help

The installation program for your Microsoft language is the recommended method for configuring on-line Help. The procedure varies, depending on whether you use OS/2 (supporting protected mode and real mode) or DOS (real mode only).

1. If you are running under DOS or real-mode OS/2 only, copy the M.HLP and MHELP.MXT files to any directory specified in the PATH environment variable in your AUTOEXEC.BAT file (DOS) or STARTUP.CMD file (OS/2).
2. If you are running under protected-mode OS/2 only, copy the M.HLP and MHELP.PXT files to any directory specified in the PATH environment variable. Also copy MSHELP.DLL to any directory listed in the LIBPATH variable in the CONFIG.SYS file. MHELP.PXT is an extension to the editor. MSHELP.DLL is a support library that implements the standard Microsoft Help engine.
3. If you are setting up for both real- and protected-mode OS/2, perform both the preceding steps.

Other Microsoft products include .HLP files that the Microsoft Editor can read. If you want to add additional .HLP files to Help, you need to include the following tagged section in your TOOLS.INI file:

```
[M-MHELP MEP-MHELP]
helpfiles:path\file.hlp
```

in which *path* is the directory of the Help file. The tag [M-MHELP MEP-MHELP] is the heading for the section in the TOOLS.INI file executed whenever Help is loaded. If this section is missing, the editor looks for the file M.HLP in your PATH directories. To install more than one Help file, you must have this tagged section in TOOLS.INI.

Whenever you use context-sensitive Help, the editor searches the files specified in the **helpfiles** switch in the order listed. The search is slower if the topics you most often inquire about are in files at the end of the list. You can alter the **helpfiles** switch at any time with the *Assign* command.

7.6.1 Controlling Search Order

Context-sensitive Help searches all the Help files for the selected topic. The order in which files are searched is important. Two different Help files may have an entry for the same topic. Furthermore, searching the appropriate file first speeds up Help.

You can use the file extension of the current file to change the order in which .HLP files are searched. Precede files specified in the **helpfiles** switch by a file extension and a colon (:):

```
helpfiles: M.HLP .BAS:QB.HLP .C.H:C.HLP .ASM:OS2.HLP
```

In the example above, Help searches QB.HLP first when the current file has a .BAS extension, C.HLP when the current file has a .C or .H extension, and OS2.HLP when the current file has a .ASM extension. In all cases, all of the files listed are searched before Help concludes it cannot find a topic.

7.6.2 Default Help File Search

The editor looks for .HLP files according to the following rules:

1. If the **helpfiles** switch is set, the editor uses the files specified in this switch.
2. If **helpfiles** is not set, the editor checks next to see if the HELPFILES environment variable is set. If it is, this environment variable is evaluated the same way the **helpfiles** switch is.
3. If neither the **helpfiles** switch nor the HELPFILES environment variable is set, the editor searches for the file M.HLP. The editor finds M.HLP if this file is in a directory listed in the PATH environment variable.

CHAPTER 8

8 Programming C Extensions

C extensions provide the most powerful means of customizing the Microsoft Editor. A "C extension" is a C-language module containing new editing functions you program. Your functions can be attached to a key, given arguments, or used in macros just as standard editing functions are. The module can also define new switches. The user can adjust these switches to modify the behavior of your functions.

With C extensions, you can use the power of the C language—data structures, control-flow structures, and C operators. Furthermore, C extensions are much faster than macros because they are compiled.

NOTE *This chapter assumes you already know how to program in C. Before you read the chapter, make sure you understand the following C-language programming concepts: functions, pointers, structures, and unions. You also need to know how to compile and link a C source file.*

You can also write extensions with the Microsoft Macro Assembler if you simulate the C memory model specified in Section 8.5.1, "Compiling and Linking in Real Mode." However, this chapter is primarily addressed to C programmers.

This chapter gradually develops concepts for writing C extensions. The first time you read the chapter, read the sections in order. The chapter first describes requirements for writing C extensions and explains how C extensions work. It then explains how to create the required objects in a C extension, to program your editing functions, to compile and link your module, and to use the editor's low-level functions to read and write to files.

8.1 Requirements

To create C extensions, you need to have the following files and software present in your current directory (or directories listed in the PATH or INCLUDE environment variables, as appropriate):

- The Microsoft C Optimizing Compiler, Version 5.1 or later. You can use Version 4.0 or 5.0 of the compiler, but the files you need are provided with Version 5.1 and later. You also get these files with the Microsoft Macro Assembler, Version 5.1 and later.
- The Microsoft Overlay Linker, Version 3.60 or later; the OS/2 version of the linker; or the Microsoft Segmented-Executable Linker, Version 5.01.
- EXTHDR.OBJ (supplied with the C compiler) or EXTHDRP.OBJ (a file supplied with the C compiler for creating protected-mode extensions).
- EXT.H (supplied with the C compiler).
- SKEL.C (a template supplied with the C compiler you can replace with your own code).

You need a minimum of 220K of available memory for the editor to load a C extension at run time, plus the size of the extension itself.

8.2 How C Extensions Work

A C-extension module is similar in the following respects to an OS/2 or Windows dynamic-link library:

- There is no function called **main** in your module. Instead, you use certain names and structures the editor recognizes.
- You compile and link to create an executable file, but this executable file is separate from the "main program," M.EXE or MEP.EXE.
- The editor loads your executable file into memory at run time and uses a table-driven method for enabling your module to call functions within the editor.

Once your executable file is loaded, it resides in memory along with the editor. The editor can call your functions, and your functions can call the Microsoft Editor's low-level functions that perform input and output operations.

The following list summarizes the process of developing and using a C extension:

1. Compile a C module with a special memory-model option, then link the resulting object file to create an executable file.

   You also link in the object file EXTHDR.OBJ (or EXTHDRP.OBJ, if linking for protected mode) to the beginning of your executable file. This object file contains a special table that enables your functions to effectively call functions within the editor.

2. Start up the Microsoft Editor. Set the **load** switch to look for the executable file you created. (As discussed in Chapter 7, the **load** switch can be set in the TOOLS.INI file or manually with the *Assign* function.)

   The editor loads your executable file into memory.

3. As soon as the executable file is loaded, the Microsoft Editor calls the function **WhenLoaded**—a special function your module must define. At the same time, the editor examines the table **cmdTable**, which is an array of structures your module must declare. The editor examines this table to recognize the editing functions you created. The table contains function names and pointers to functions.

4. You can assign keys to call your functions. Assign a key manually or in the **WhenLoaded** function, then press the assigned key. You can also call an editing function indirectly by placing it in a macro and calling the macro.

5. When you invoke a C-extension function, the editor responds by calling your module.

6. Your editing function is executed. It calls the Microsoft Editor's low-level functions to read from the text file, read output to the text file, and print messages.

8.3 Writing a C Extension

To create a successful C extension, you need to follow these guidelines:

1. Include the file EXT.H.

   This file declares all structures and types required to establish an interface to the editor.

2. Include the standard items that are described in Section 8.3.1, "Required Objects," and write your functions by following the steps in Section 8.4, "Programming Your Function." Then compile and link as directed in Section 8.5, "Compiling and Linking."

3. Call the low-level extension functions to do most any operation with files, such as read from a file, write to a file, delete or add files, and move the cursor. These functions are completely described in Chapter 9, "C-Extension Functions," (an alphabetical reference), and summarized by topic at the end of this chapter.

Do not call functions from the C library routine, except for the ones specifically listed in Section 8.7, "Calling Library Functions." Futhermore, floating-point arithmetic is not supported.

8.3.1 Required Objects

A C-extension module must have at minimum the three objects described below:

Object Name	Description
swiTable	An array of structures that declares internal switches you wish to create
cmdTable	An array of structures that declares editing functions you have coded
WhenLoaded	A function that the editor calls as soon as the C-extension module is loaded

Each of these items can be as short or long as you wish. Each table can be as short as a single row of entries. The **WhenLoaded** function can return immediately, or it can perform useful initialization tasks, such as assigning keys to functions or printing a message.

8.3.2 The Switch Table

The switch table, **swiTable**, consists of a series of structures in which each structure describes a switch you wish to create. The table ends with a structure that has all null (all zero) values. Though you may choose not to create any switches, the table must still be present. The simplest table allowed is therefore

```
struct swiDesc swiTable[] =
{
     { (void *) 0, (void *) 0, 0 }
};
```

The structure type **swiDesc** is defined in EXT.H. This structure contains the following three fields that define a switch for the editor to recognize:

1. A pointer to the name of the switch.
2. A pointer to the switch itself or to a function. If the switch is Boolean, this field must point to the switch (an integer which assumes the value –1 or 0). If the switch is text, this field must point to a function, as explained below. If the switch is numeric, this field points to switch itself—an integer.
3. A flag that indicates the type of switch: either **SWI_BOOLEAN**, **SWI_NUMERIC**, or **SWI_SPECIAL**.

If the third field has value **SWI_NUMERIC**, you must combine it with the value **RADIX10** or **RADIX16** by using binary or (|). (See the second example below.) If you specify **RADIX10**, the editor interprets user-assigned values as decimal digits. If you specify **RADIX16**, the editor interprets these values as hexadecimal.

If the third field has value **SWI_SPECIAL**, the second field must be a pointer to a function of type **int far pascal**. You define this function in your code. Each time the value of the switch changes, the editor calls your function and passes the updated value in a character string. Your function should declare exactly one parameter: a far pointer to a character string. Typically, this function might set a global variable:

```
char globalstr[BUFLEN];

int far pascal setstr (char far *p)
{
        strcpy (globalstr, p);
}
```

The table may have any number of rows (each row being a structure), and must at least include the final row of all null values. Here is an example of a table that creates a numeric switch with a default value of 27:

```
int n = 27;

struct swiDesc swiTable [] =
{
     { "newswitch", &n, SWI_NUMERIC|RADIX10  } ,
     { (void *) 0, (void *) 0, 0 }
}
```

8.3.3 The Command Table

The command table, **cmdTable**, is similar to the switch table, **swiTable**, in its construction. Each "row" of the table consists of a structure that describes an editing function you want the editor to recognize. The last row must contain all null values. The simplest table allowed is the following:

```
struct cmdDesc cmdTable[] =
{
     { (void *) 0, (void *) 0, 0, 0 }
}
```

Usually you'll want to declare at least one new editing function. The structure type **cmdDesc** is defined in EXT.H. This structure contains the following four fields that make an editing function recognizable to the editor:

1. A pointer to the name of the function, in ASCII format. This name could appear in assignments and macros.

2. The address of the function itself. Give the function name but do not follow it with parentheses.
3. A field used internally by the editor. Always declare this field as null.
4. Flags indicating the type of the function. Function types are described below and define what type of argument the function will accept.

Here is an example of a command table declaring a function that takes no arguments and a second function that takes either a *linearg* or a *boxarg* or can be entered without arguments:

```
struct cmdDesc cmdTable[] =
{
     { "newfun", newfun, 0, NOARG } ,
     { "fun2",   fun2,   0, LINEARG | BOXARG | NOARG } ,
     { (void *) 0, (void *) 0, 0, 0 }
}
```

In the fourth field of the command table, use one or more of the values described in Table 8.1 below.

The first column of Table 8.1 contains a flag you can enter in the fourth field of **cmdTable**. The second column describes the associated behavior. The third column indicates which of the six data formats the editor uses to pass information. Your function determines the format at run time by testing the value of pArg->argType. Sections 8.4.3–8.4.8 give complete descriptions of these formats.

Many of the descriptions below state that the editor responds by passing certain information. However, these responses only apply when the editor detects the associated condition. For example, the **BOXARG** flag controls the editor's behavior only when a *boxarg* is detected. However, **KEEPMETA**, **MODIFIES**, and **CURSORFUNC** apply regardless of argument type.

Table 8.1 Meaning of cmdTable Flags

cmdTable Flag	Behavior	Resulting argType Format
KEEPMETA	Preserves the state of the *Meta* prefix, which is normally reset upon completion of the function.	Not applicable
MODIFIES	Prevents execution of the function when the current file is No-Edit.	Not applicable
WINDOWFUNC	Does not cancel highlight resulting from a previous command, such as *Psearch*. This flag is useful for window-movement functions.	Not applicable

Table 8.1 *(continued)*

cmdTable Flag	Behavior	Resulting argType Format
CURSORFUNC	Does not recognize or cancel *Arg* prefix. The function is useful for helping define cursor-movement arguments, since it preserves the preexisting argument. Conflicts with all flags except **KEEPMETA** and **MODIFIES**.	**NOARG**
NOARG	Accepts absence of the *Arg* prefix. The editor passes location of the cursor.	**NOARG**
NULLARG	Accepts *Arg* without an argument. The editor passes location of the cursor. Conflicts with **NULLEOL** and **NULLEOW**.	**NULLARG**
NULLEOL	Accepts *Arg* without an argument. The editor passes the string of text from the cursor position up to the end of the line. Conflicts with **NULLARG** and **NULLEOW**.	**TEXTARG**
NULLEOW	Accepts *Arg* without an argument. The editor passes the string of text from the cursor position up to the next white space. Conflicts with **NULLARG** and **NULLEOL**.	**TEXTARG**
TEXTARG	Accepts a text argument that the user types in directly.	**TEXTARG**
BOXSTR	Accepts a one-line box argument. The editor passes the highlighted string as a text argument.	**TEXTARG**
LINEARG	Accepts a *linearg*. The editor passes location of first and last lines.	**LINEARG**
BOXARG	Accepts a *boxarg*. The editor passes location of the four edges of the box.	**BOXARG**
STREAMARG	Interprets any cursor-movement argument as a stream of text. The stream may span more than one line. The editor passes location of beginning and ending points.	**STREAMARG**
NUMARG	Interprets a numeric text argument as a file position exactly *numarg* lines down from the cursor. This position and the cursor position then define a region in the file. (The two positions form the end points of the region.) Use this flag in conjunction with other cursor-movement-argument flags, especially **LINEARG**.	Any cursor-movement type
MARKARG	Interprets a valid file-marker name as a file position. This position and the cursor position then define a region in the file. (The two positions form the end points of the region.) Use this flag in conjunction with other cursor-movement-argument flags.	Any cursor-movement type

The descriptions also refer to the passing of information to the function; you'll see how the function receives information in Section 8.3.5, "Defining the Editing Function."

The flags are bit masks; each turns a specific bit within an unsigned integer. You can combine these bit masks with binary or (|). For example, you can specify a function that accepts a *boxarg*, *linearg*, or *numarg* as:

```
BOXARG | LINEARG | NUMARG
```

8.3.4 The WhenLoaded Function

The function **WhenLoaded** takes no arguments and can return immediately. However, you must include the function because the editor expects it to be present. The simplest version of **WhenLoaded** is this:

```
WhenLoaded()
{
        return TRUE;
}
```

In Section 8.7, "Calling Library Functions," you'll learn how to call functions that assign keys to functions and print a message on the message line. These functions are often useful to call from within **WhenLoaded**.

8.3.5 Defining the Editing Function

This section describes how to define an editing function. The section also gives an overview of the information the editor passes to every function. Section 8.4, "Programming Your Function," gives specific information on how to interpret user-defined arguments.

The editing function must be declared **flagType pascal EXPORT**. The **flagType** declaration indicates that your function returns one of two values: TRUE (non-zero) or FALSE (zero). (Returning a value is recommended, because once loaded, your function can be used within a macro. However, returning a value is not a strict requirement.) The **pascal** keyword indicates that your function uses the Pascal calling convention. Finally, your function must be of type **EXPORT** to be properly accessed through the editor's dynamic-link calls.

The sample function `Skel` is declared as follows:

```
#define TRUE  -1
#define FALSE 0

flagType pascal EXPORT Skel (argData, pArg, fMeta)
unsigned int argData;
ARG far *pArg;
flagType fMeta;
{
        return TRUE;
}
```

Replace the name `Skel` by the name of your function. This same name should appear in the second field of the **cmdTable** data structure.

The parameter list is described below:

Parameter	Description
argData	The value of the keystroke used to invoke the function. This parameter is generally not used.
pArg	A pointer to a structure that contains almost all the information passed by the editor. This structure is discussed in detail below.
fMeta	An integer that describes whether or not a *Meta* prefix is present. This integer has value true (nonzero) if *Meta* is present and value false (0) if not.

The parameter **pArg** points to a structure whose first member is **argType**. This variable contains one of six values: **NOARG**, **TEXTARG**, **NULLARG**, **LINEARG**, **STREAMARG**, or **TEXTARG**. Each of these values is defined in Section 8.3.3, "The Command Table." For example, you could test for the presence of a *boxarg* with the following code:

```
if( pArg->argType == BCXARG )
{
    /* take appropriate action for boxarg */
}
```

The rest of the structure consists of a union of smaller structures. The C **union** type is necessary here; it enables the editor to pass data in a variety of formats. The choice of format depends on what kind of argument the user defined.

The declaration of the **ARG** structure is:

```
struct argType
{
    int   argType;
    union
    {
        struct  noargType       noarg;
        struct  textargType     textarg;
        struct  nullargType     nullarg;
        struct  lineargType     linearg;
        struct  streamargType   streamarg;
        struct  boxargType      boxarg;
    } arg;
}

typedef struct argType ARG;
```

The editor returns argument information in one of the nested structures. The next section describes the contents of each of these structures and provides examples showing how to use each data format.

8.4 Programming Your Function

When you write an editing function, you have two basic tasks. First, you interpret information that the editor passes you. Second, you call the editor's lower-level functions to execute basic editing tasks.

This section focuses on the most typical class of editing functions: functions that analyze the user-defined argument and modify only the current file. This class of functions, although huge in scope, depends on only a few lower-level functions. Once you learn how to write these functions, you can progress to more advanced operations—such as working on several files at once or reading keyboard input. Chapter 9 describes the whole range of low-level functions available.

A typical editing function might execute the following sequence of steps:

1. Get a handle to the current file by calling **FileNameToHandle**.
2. Use information about the user-defined argument to initialize local variables.
3. Process the argument. Modify the current file by calling **GetLine** and **Putline**.
4. Return TRUE or FALSE.

The order above is recommended, but you do not have to follow it strictly. You can perform Step 2 before Step 1. However, you must get a handle to the current file before you can modify it.

Sections 8.4.1–8.4.9 discuss each step.

8.4.1 Getting a File Handle

The Microsoft Editor has its own file system. A "file system" is a group of integrated structures and function calls for modifying files. The editor's file system is not compatible with that of the C library.

The editor has its own file-handle type, **PFILE**. This type, along with other types needed for extensions, is defined in the include file EXT.H. For example, you can create a file handle `pFile` with the declaration:

```
PFILE   pFile;
```

Usually, your editing work involves the current file; that is, the file that now appears in the current editing window. Because the current file is already open for editing, you do not need to open or initialize the current file in any way. You simply assign the existing file handle to one of your own variables.

The **FileNameToHandle** function returns a handle to a file that is already open for editing. (More than one file may be open for editing, particularly if the user has worked on more than one file during the editing session.)

To get a handle to the current file, simply pass two empty strings to the function:

```
pFile = FileNameToHandle( "", "" );
```

The first argument to **FileNameToHandle** takes a file name. The second argument takes a "short name," which is a base name the editor searches for in its list of open files. If the first string is empty, the function returns a handle to the current file.

The **AddFile** function is the other function that returns a file handle. **AddFile** causes the editor to open a file for editing—either a new file or a file that exists on disk but is not yet open for editing.

See Chapter 9 for complete information on each function.

8.4.2 Interpreting the User-Defined Argument

After getting a handle to the current file, you need to interpret the user-defined argument.

You determine the type of the user's argument by testing the value of pArg->argType. If you declare a function that takes only one type of argument, testing pArg->argType is not necessary. You know in advance what kind of argument was given, since the editor rejects invalid arguments. Table 8.1, "Meaning of cmdTable Flags," explains how to declare what arguments your function accepts.

If your function accepts more than one kind of argument, you can use either if-then-else blocks or switch-case statements to evaluate the user's argument:

```
switch( pArg->argType )
{
   case( NULLARG ):     /* Take action for empty arg  */
   .
   .
   .
   case( BOXARG ):      /* Take action for boxarg */
   .
   .
   .
   case( LINEARG ):    /* Take action for linearg */
   .
   .
   .
}
```

The value of pArg->argType is always equal to one of six values:

- **NOARG**
- **NULLARG**
- **TEXTARG**
- **LINEARG**
- **STREAMARG**
- **BOXARG**

The next six sections consider each of these argument types in detail, giving declarations and examples for each.

8.4.3 The NOARG Type

When pArg->argType is equal to **NOARG**, the editor passes information in the pArg->arg.noarg structure, which has the format:

```
struct  noargType
{
    LINE    y;    /* Line number of cursor */
    COL     x;    /* Column of cursor */
};
```

Description

The **NOARG** type indicates that the user did not invoke the *Arg* prefix, and therefore did not enter an argument. The cursor coordinates are zero-based; in other words, the start of the file is position (0,0).

This argument type is possible if you declared your function with the flag **NOARG** or **CURSORFUNC**.

Example

The following example initializes the coordinates (`yCur, xCur`) with the row and column of the cursor position:

```
COL   xCur;
LINE  yCur;
.
.
.
if( pArg->argType == NOARG )
{
        xCur = pArg->arg.noarg.x;
        yCur = pArg->arg.noarg.y;
}
```

8.4.4 The NULLARG Type

When pArg->argType is equal to **NULLARG**, the editor passes information in the pArg->arg.nullarg structure, which has the format:

```
struct  nullargType
{
    int     cArg;       /* Number of times Arg was invoked */
    LINE    y;          /* Line number of cursor */
    COL     x;          /* Column of cursor */
};
```

Description

The **NULLARG** type indicates that the user invoked the *Arg* prefix but did not enter an argument. The argument is therefore empty, or null. The cursor coordinates are zero-based; in other words, the start of the file is position (0,0).

This argument type is possible if you declared your function with the flag **NULLARG**.

Example

The following example initializes the coordinates (`yCur, xCur`) with the row and column of the cursor position, and sets `cArg` equal to the number of times that the user invoked the *Arg* prefix:

```
COL   xCur;
LINE  yCur;
int   cArg;
.
.
.
if( pArg->argType == NULLARG )
{
        xCur = pArg->arg.nullarg.x;
        yCur = pArg->arg.nullarg.y;
        cArg = pArg->arg.nullarg.cArg;
}
```

8.4.5 The TEXTARG Type

When pArg->argType is equal to **TEXTARG**, the editor passes information in the pArg->arg.textarg structure, which has the format:

```
struct  textargType
{
    int      cArg;        /* Number of times Arg was invoked */
    LINE     y;           /* Line number of cursor */
    COL      x;           /* Column of cursor */
    char far *pText       /* Pointer to textarg string */
};
```

Description

The **TEXTARG** type indicates that the user's argument defined a string of text. The cursor coordinates are zero-based, and `pText` points to a null-terminated string.

This argument type is possible if you declared your function with the flag **TEXTARG**, **BOXSTR**, **NULLEOL**, or **NULLEOW**. Each of these argument types selects a string of text in a different way (for example, **BOXSTR** uses a cursor-movement argument); however, the format used for all these cases is the same.

Example

The following example initializes the coordinates (`yCur, xCur`) with the row and column of the cursor position, and sets `cArg` equal to the number of times that the user invoked the *Arg* prefix. Finally, the user-defined *textarg* is copied into the character array `buf`.

```
COL   xCur;
LINE  yCur;
int   cArg;
char  buf[BUFLEN]; .
.
.
if( pArg->argType == TEXTARG )
{
        xCur = pArg->arg.textarg.x;
        yCur = pArg->arg.textarg.y;
        cArg = pArg->arg.textarg.cArg;
        strcpy (buf, pArg->arg.textarg.pText);
}
```

8.4.6 The LINEARG Type

When pArg->argType is equal to **LINEARG**, the editor passes information in the pArg->arg.linearg structure, which has the format:

```
struct  lineargType
{
    int     cArg;       /* Number of times Arg was invoked */
    LINE    yStart;     /* Line number of first line */
    LINE    yEnd;       /* Line number of last line */
};
```

Description

The **LINEARG** type indicates that the user defined a range of lines by aligning the cursor and arg position in the same column but different rows. The range of lines includes `yStart` and `yEnd`, as well as all lines in between. Line numbers are zero-based, so that line 0 is the first line in the file.

This argument type is possible if you declared your function with the flag **LINEARG**.

Example

The following example sets `cArg` equal to the number of times that the user invoked the *Arg* prefix, and sets `yStart` and `yEnd` equal to the line numbers of the first and last lines:

```
int   cArg;
LINE  yStart;
LINE  yEnd;
.
.
.
if( pArg->argType == LINEARG )
{
        cArg   = pArg->arg.linearg.cArg;
        yStart = pArg->arg.linearg.yStart;
        yEnd   = pArg->arg.linearg.yEnd;
}
```

8.4.7 The STREAMARG Type

When pArg->argType is equal to **STREAMARG**, the editor passes information in the pArg->arg.linearg structure, which has the format:

```
struct streamargType
{
    int   cArg;          /* Number of times Arg was invoked */
    LINE  yStart;        /* Coordinates of first byte in    */
    COL   xStart;        /*   the stream of text            */
    LINE  yEnd;          /* Coordinates of byte just after  */
    COL   xEnd;          /*   the last byte of the stream   */
};
```

Description

The **STREAMARG** type indicates that the user defined a cursor-movement argument, which the function interprets as a stream of text rather than as a *boxarg* or *linearg*. The stream of text includes `yStart` and `xStart` but not `yEnd` and `xEnd`, which gives the location of the character just to the right of the last character in the stream.

This argument type is possible if you declared your function with the flag **STREAMARG**.

Example

The following example sets `cArg` equal to the number of times that the user invoked the *Arg* prefix. Then the example initializes variables for the beginning and end of the stream.

```
int   cArg;
LINE  yStart;
COL   xStart;
LINE  yEnd;
COL   xEnd;
.
.
.
if( pArg-argType == LINEARG )
{
        cArg   = pArg->arg.streamarg.cArg;
        yStart = pArg->arg.streamarg.yStart;
        xStart = pArg->arg.streamarg.xStart;
        yEnd   = pArg->arg.streamarg.yEnd;
        xEnd   = pArg->arg.streamarg.xEnd;
}
```

8.4.8 The BOXARG Type

When pArg->argType is equal to **BOXARG**, the editor passes information in the pArg->arg.boxarg structure, which has the format:

```
struct  boxargType
{
    int     cArg;       /* Number of times Arg was invoked */
    LINE    yTop;       /* Line number of first line */
    LINE    yBottom;    /* Line number of last line */
    COL     xLeft;      /* Leftmost column in box */
    COL     xRight;     /* Rightmost column in box */
};
```

Description

The **BOXARG** type indicates that the user defined a rectangular area on the screen. Row and column numbers are zero-based; in other words, the start of the file is position (0,0). The lines and column edges, `yTop`, `yBottom`, `xLeft`, and `xRight`, are all included in the area itself.

This argument type is possible if you declared your function with the flag **BOXARG**.

Example

The following example sets `cArg` equal to the number of times that the user invoked the *Arg* prefix. Then the example initializes variables for the four borders of the box.

```
int   cArg;
LINE  yTop;
LINE  yBottom;
COL   xLeft;
COL   xRight;
.
.
.
if( pArg-argType == LINEARG )
{
        cArg    = pArg->arg.boxarg.cArg;
        yTop    = pArg->arg.boxarg.yTop;
        yBottom = pArg->arg.boxarg.yBottom;
        xLeft   = pArg->arg.boxarg.xLeft;
        xRight  = pArg->arg.boxarg.xRight;
}
```

8.4.9 Modifying the Current File

This section deals with the core of an editing function—reading and altering the file.

After you have analyzed the user's argument and got a handle to the current file, you are ready to work on the file. At this stage, two functions are most relevant: **GetLine** and **Putline**.

The **GetLine** function reads one line from the file:

```
len = GetLine (yStart, buffer, pFile);
```

The first argument is a line number, the second a pointer to a buffer, and the third a file handle. With extension functions, line numbers are consistently indexed (they are all zero-based so that the first line in the file is numbered 0), so you can use a line number previously passed by the editor. The buffer is a previously declared string of characters. **GetLine** fills this string with characters and a terminating null byte. Since you are working on the current file, pass the file handle you received when you called **FileNameToHandle**.

Finally, **GetLine** returns the length of the line copied.

The **PutLine** function is the flip side of **GetLine** and takes almost exactly the same arguments:

```
PutLine (yStart, buffer, pFile);
```

The first argument is a zero-based line number. The second argument points to a buffer containing text that you want to write to the file. The third argument is a file handle. Again, you can use a handle to the current file.

Both **GetLine** and **PutLine** deal with null-terminated strings. You should not add a new-line character to the string you pass to **PutLine**. The editor adds or strips new-line characters as appropriate.

The following section of code demonstrates the use of **GetLine** and **PutLine** in a loop that numbers the lines `yStart` to `yEnd`:

```
for( curLine = yStart, i = 1; curLine <= yEnd; curLine++, i++ )
{
    GetLine( curLine, buffer, pFile );    /* Get line */

    itoa( i, digits, 10 );                /* Convert i to digit string */
    strcat( digits, ". " );
    strcat( digits, buffer );             /* Prefix buffer with digits */

    PutLine( curLine, digits, pFile );    /* Put line back */
}
```

8.5 Compiling and Linking

After writing your C module, you're ready to compile and link. The procedures for compiling and linking in protected mode are slightly different from compiling and linking in real mode. Sections 8.5.1 and 8.5.2 consider both environments.

NOTE *If you created C-extension modules for Version 1.0 of the Microsoft Editor, you do not have to recompile or relink your C extensions. However, if you do recompile your extensions, make sure that you use the new version of the EXT.H include file.*

8.5.1 Compiling and Linking for Real Mode

To create a C extension for real mode (DOS or the OS/2 3. *x* compability box), follow these steps:

1. Compile with command line options /Gs and /Asfu. These mandatory options establish the proper memory model and calling convention. (If you are programming in the Microsoft Macro Assembler (MASM), use near code and far data segments, in which SS is not assumed equal to DS.) For example:

   ```
   CL /c /Gs /Asfu myext.c
   ```

2. Link the file EXTHDR.OBJ to your extension. The EXTHDR.OBJ file must be the first object module on the command line:

   ```
   CL /AC /Femyext.mxt exthdr myext
   ```

Use the /AC option if your extension calls C library functions. The /AC option directs the linker to use the compact-model library.

IMPORTANT *Strictly speaking, the .MXT file extension is not required; the editor can load modules with any file extension name. However, using a file extension of .MXT is strongly recommended so that your extensions are not confused with true executable (.EXE) files, which can be run directly from DOS. Trying to execute an extension as if it were an .EXE file will bring the system to its knees.*

8.5.2 Compiling and Linking for Protected Mode

To create a C extension for protected mode, follow these steps:

1. Compile with command-line options /Gs and /Asfu. These mandatory options establish the proper memory model and calling convention. (If you are programming in MASM, use near code and far data segments in which SS is not assumed equal to DS.) For example:

   ```
   CL /c /Gs /Asfu myext.c
   ```

   The same object module can be used for both real and protected mode, so you do not have to recompile if you want an extension for both modes. However, you do have to relink, as explained below.

2. Link with the file EXTHDRP.OBJ to your extension along with a module-definitions file. The EXTHDRP.OBJ file must be the first object module on the command line:

   ```
   LINK /NOI exthdrp myext, myext.dll,, doscalls, myext.def;
   ```

   If your extension calls C library functions, link in the compact-model library. Link with the /NOI option, since the OS/2 loader is case-sensitive.

The file MYEXT.DEF can be created by copying the file SKEL.DEF (provided with the editor). No modification is necessary. You can also link with SKEL.DEF itself.

8.5.3 Loading Your Extension

Compiling and linking your module with the correct options produces an .MXT file (for real mode) or a .DLL file (for OS/2 protected mode) the editor can load and execute on demand.

To use a real-mode extension, set the **load** switch with the full path name for your C extension. For example, after you have created the file MYEXT.MXT, you could place the following statement in the TOOLS.INI file:

```
load:path\myext.mxt
```

in which *path* is the location of the extension. The editor responds by automatically loading your C-extension module into memory upon start-up. You can also set the **load** switch with the *Assign* command.

NOTE *The load switch can be assigned a number of times, each assignment causing the editor to load another extension. All of the extensions reside in memory together; no extension is removed from memory until the editor terminates.*

The load switch can accept an environment variable as described in Section 7.3, "Special Syntax for Text Switches."

To use a protected-mode extension, place your .DLL file in a directory specified in the LIBPATH variable. You can set the load switch in the same way described above; however, when running under protected mode, the editor ignores the *path* and the file extension. Therefore, for protected-mode-only extensions, the only declaration you need is the following:

```
load:myext
```

To load an extension for both real and protected modes, follow these steps:

1. Compile the source file once.
2. Link two times, once for real mode and once for protected mode.
3. Place the protected-mode module (a .DLL file) in a LIBPATH directory.
4. Use the real-mode **load** setting. The editor uses the full text of the switch setting when in real mode, but ignores the path and file extension when in protected mode.

Whenever the editor successfully loads an extension, it checks the TOOLS.INI file for the tag

[M-*mxt*]

in which *mxt* is the base name of the extension. If the tag exists, the editor recognizes the settings immediately following the tag.

Special Considerations for OS/2

Search order with the load statement

Under OS/2 Version 1.1 and later, protected-mode editor extensions may have file extensions other than .DLL and may be placed in directories other than those specified in LIBPATH. The editor searches for the extensions specified in the **load** statement in the following sequence:

1. If an environment variable or a full path is specified, the editor searches the appropriate directories, as in the following examples:

   ```
   load:$LIB:yourext.mxt
   ```

   ```
   load:drive:\dir\yourext.mxt
   ```

   The editor searches the current directory if there is no environment variable or full-path specification.

2. If the desired extension is not found, and its name is only a base name without a file extension, the editor appends the appropriate file extension (.MXT for real mode, .PXT for protected mode) and repeats Step 1.

3. If the editor extension still cannot be located, the editor searches for *basename*.DLL in the directories specified in LIBPATH, where *basename* is the base name of the extension.

In real mode (DOS or the OS/2 3.*x* compatibility box), only Steps 1 and 2 are preformed. Under OS/2 Version 1.0, only Step 3 is performed.

Within TOOLS.INI tag declarations, only the base name is significant, not the file extension. For example, the statements following

```
[M-YOUREXT]
```

would always be used, regardless of the file extension that was used to locate YOUREXT.

The "autoload" feature automatically loads extensions without a **load** statement in TOOLS.INI. The editor searches the directories in the PATH environment variable for file names with the following patterns, and loads as many of them as it finds:

Operating System	Extension Autoloaded
DOS or OS/2 real mode	M*.MXT
OS/2 protected mode	M*.PXT

The file names for the on-line Help extensions supplied with this version of the Microsoft Editor have the correct form for autoloading. Therefore, they only

need to be placed in any directory in the PATH environment variable; you do not have to add a **load** statement to TOOLS.INI.

Autoloading occurs after TOOLS.INI has been read, so any extension-specific sections still take effect.

NOTE *Autoloading does not work under OS/2 Version 1.0.*

You can load extensions in both real and protected mode by using a single **load** statement of the following form:

`load:$PATH:`*`basename`*

The extensions must have the same *basename* and the default extension (.MXT for DOS or OS/2 real mode, .PXT for OS/2 protected mode) for the appropriate operating system. The extensions can go anywhere in the specified path. Since the file extension is omitted in this **load** statement, the editor will append the appropriate extension to *basename* (be sure the name has no trailing period) and load the correct file.

You can also give your extension a name of the form M*.MXT or M*.PXT (according to the operating system). The extension will be autoloaded without requiring a **load** statement in TOOLS.INI. It is also possible to specify the editor extension for one mode in a **load** statement, while giving the extension for the other mode an autoload default name.

8.6 A C-Extension Sample Program

The following C-extension sample program features a function named `tglcase`. This function converts uppercase letters to lowercase, and lowercase letters to uppercase. It acts on the file according to the argument type:

Argument	Effect
NOARG	Acts on entire current line.
NULLARG	Acts on current line, from the cursor position (inclusive) up to the end of the line.
BOXARG	Acts on the highlighted region.
LINEARG	Acts on the highlighted region (containing a range of complete lines).
TEXTARG	Argument not accepted.
STREAMARG	Not recognized. The argument is treated as **BOXARG** or **LINEARG**.

In each case, the function responds to the argument by initializing the limits of an imaginary box. In the case of a **LINEARG**, the column limits are set at 0 and BUFLEN (the maximum length of a text line in M). **NOARG** and **NULLARG** are handled in a similar fashion. Once the limits of the area are initialized, one small section of code does the actual work. This section calls **GetLine** and **PutLine** repeatedly to read and replace lines in the current file.

This extension calls C library functions. To successfully link this extension, include the compact-model library.

```
/*** tglcase.c - case toggling editor extension
*/
#define ID      " tglcase ver 1.00 "##__DATE__##" "##__TIME__
#define NULL    ((void *) 0)

#include <stdlib.h>                     /* min macro definition         */
#include <string.h>                     /* prototypes for string fcns   */
#include "ext.h"
/*
** Internal function prototypes
*/
void     pascal          id          (char *);
void     WhenLoaded (void);
flagType pascal EXPORT tglcase    (unsigned int, ARG far *, flagType);

/*****************************************************************************
**
** tglcase
** Toggle the case of alphabetics contained within the selected argument:
**
**  NOARG       - Toggle case of entire current line
**  NULLARG     - Toggle case of current line from cursor to end of line
**  LINEARG     - Toggle case of range of lines
**  BOXARG      - Toggle case of characters with the selected box
**  NUMARG      - Converted to LINEARG before extension is called
**  MARKARG     - Converted to Appropriate ARG form above before extension is
**                called
**
**  STREAMARG   - Not Allowed; treated as BOXARG
**  TEXTARG     - Not Allowed
**
*/
flagType pascal EXTERNAL tglcase (argData, pArg, fMeta)
unsigned int argData;                   /* keystroke invoked with       */
ARG far *pArg;                          /* argument data                */
flagType fMeta;                         /* indicates preceded by meta   */
{
    PFILE   pFile;                    /* file handle of current file  */
    COL     xStart;                   /* left border of arg area      */
    LINE    yStart;                   /* starting line of arg area    */
    COL     xEnd;                     /* right border of arg area     */
    LINE    yEnd;                     /* ending line of arg area      */
    int     cbLine;                   /* byte count of current line   */
    COL     xCur;                     /* current column being toggled */
    char    buf[BUFLEN];              /* buffer for line being toggled*/
    register char c;                  /* character being analyzed     */
```

```
    id( "" );
    pFile = FileNameToHandle( "", NULL );
    switch( pArg->argType )
    {
/*
** For the various argument types, set up a box
** (xStart, yStart) - (xEnd, yEnd)
** over which the case conversion code below can operate
*/
        case NOARG:                         /* case switch entire line      */
            xStart = 0;
            xEnd = BUFLEN;
            yStart = yEnd = pArg->arg.noarg.y;
            break;

        case NULLARG:                       /* case switch to EOL           */
            xStart = pArg->arg.nullarg.x;
            xEnd = BUFLEN;
            yStart = yEnd = pArg->arg.nullarg.y;
            break;

        case LINEARG:                       /* case switch line range       */
            xStart = 0;
            xEnd = BUFLEN;
            yStart = pArg->arg.linearg.yStart;
            yEnd = pArg->arg.linearg.yEnd;
            break;

        case BOXARG:                        /* case switch box              */
            xStart = pArg->arg.boxarg.xLeft;
            xEnd   = pArg->arg.boxarg.xRight;
            yStart = pArg->arg.boxarg.yTop;
            yEnd   = pArg->arg.boxarg.yBottom;
            break;
    }
/*
** Within range of lines yStart to yEnd, get each line, and if non-null,
** check each character. If alphabetic, replace with its case-converted
** value. After all characters have been checked, replace line in file.
*/
    while( yStart <= yEnd )
    {
        if( cbLine = GetLine( yStart, buf, pFile ))
        {
            for( xCur = xStart; (xCur <= min( cbLine, xEnd )); xCur++ )
            {
                c = buf[xCur];
                if( (c >= 'A' ) && (c <= 'Z'))
                    c += 'a'-'A';
                else if( (c >= 'a' ) && (c <= 'z'))
                    c += 'A'-'a';
                buf[xCur] = c;
            }
            PutLine( yStart++, buf, pFile );
        }
    }
    return 1;
}
```

```
/*************************************************************************
**
** WhenLoaded
** Executed when extension gets loaded. Identify self & assign default
** keystroke.
**
** Entry:
**  none
*/
void WhenLoaded ()
{
id("case conversion extension:");
SetKey ("tglcase",  "alt+c");
} /* end WhenLoaded */

/*************************************************************************
**
** id
** identify ourselves, along with any passed informative message
**
** Entry:
**  pszMsg      = Pointer to asciiz message to which the extension name
**                and version are appended prior to display
*/
void pascal id (pszFcn)
char *pszFcn;                                   /* function name        */
{
    char    buf[80];                                /* message buffer       */

    strcpy (buf,pszFcn);                    /* start with message   */
    strcat (buf,ID);                            /* append version       */
    DoMessage (buf);
} /* end id */


/*************************************************************************
**
** Switch communication table to the editor
** This extension defines no switches
*/
struct swiDesc  swiTable[] =
{
    { NULL, NULL, 0 }
};

/*************************************************************************
**
** Command communication table to the editor
** Defines the name, location and acceptable argument types
*/
struct cmdDesc  cmdTable[] =
{
    {"tglcase",tglcase,0, KEEPMETA | NOARG | BOXARG | NULLARG | LINEARG |
                          MARKARG | NUMARG | MODIFIES},
    { NULL, NULL, 0, 0 }
};
```

8.7 Calling Library Functions

This section lists compatible C library functions as well as low-level extension functions you can call.

You should call the editor's own low-level functions in preference to functions in the standard C library whenever possible. Using the editor's low-level functions guarantees that all file operations are compatible with the editor.

Furthermore, not all functions in the C library are compatible with extensions. The following list summarizes which functions from the compact-model library should work when called by a C-extension module. Link with a compact-memory-model C library if you want to call these functions. The list refers to the function categories from Chapter 4 of the *Microsoft C Optimizing Compiler Run-Time Library Reference* (Version 4.0 or later).

Note that floating-point arithmetic is not supported because it involves calls to low-level, floating-point math routines.

Category	Compatible Functions
Buffer manipulation	All functions can be called.
Character classification and conversion	All functions can be called.
Data conversion	All functions can be called except **strtod**.
Directory control	All functions can be called except **getcwd**.
Graphics	None.
File handling	All functions can be called.
Stream routines	None.
Low-level I/O routines	None.
Console and port I/O	All functions can be called except **cgets**, **cprintf**, and **cscanf**.
Math	None.
Memory allocation	None.
Process control	None.
Searching and sorting	All functions can be called except **qsort**.
String manipulation	All functions can be called except **strdup**.

BIOS interface	All functions can be called.
MS-DOS interface	All functions can be called except **int86** and **int86x**.
Time	None.
Miscellaneous	All functions can be called except **assert**, **getenv**, **perror**, **putenv**, and **_searchenv**.

Table 8.2 lists the low-level extension functions by category. Your extension can call any of these functions. The next chapter gives an alphabetical reference to all of the functions, including declarations and examples.

Table 8.2 Summary of Extension Functions by Category

Category	Functions	Description
File Handle	**AddFile**	Opens new file and gets file handle
	FileNameToHandle	Gets handle to already opened file
	Remove File	Removes file structure from memory
Line-Oriented	**FileLength**	Returns number of lines in file
	GetLine	Gets contents of one line
	PutLine	Replaces a line
Cursor	**GetCursor**	Gets cursor position
	MoveCur	Moves cursor to new location
Display	**BadArg**	Reports that argument was invalid
	Display	Forces immediate update of screen
	DoMessage	Puts message on the dialog line
File-Oriented	**DelFile**	Deletes contents of a file buffer
	FileRead	Copies disk file to file buffer
	FileWrite	Copies file buffer to disk file
	pFileToTop	Makes specified file the current file
Block Operations	**CopyBox**	Inserts rectangular area
	CopyLine	Inserts range of lines
	CopyStream	Inserts stream of text
	DelBox	Deletes rectangular area
	DelLine	Deletes range of lines
	DelStream	Deletes stream of text

Table 8.2 *(continued)*

Category	Functions	Description
Keyboard	**KbHook**	Restores keyboard control to M
	KbUnHook	Removes keyboard control from M
	ReadChar	Returns information on next keystroke
	ReadCmd	Returns keystroke info in **CmdDesc** format
Miscellaneous	**fExecute**	Executes a macro
	Replace	Replaces one character in a file
	SetKey	Assigns a function to a keystroke

CHAPTER 9

C-Extension Functions

9

Most of the real work of an extension is done by the editor. Your extension provides program logic and decision making. Yet it relies on the editor to interact with the environment. Specifically, an extension calls low-level functions within the editor itself to alter a file, update the screen, read keyboard input, and perform many other useful functions.

This chapter describes these low-level functions in alphabetical order. Most descriptions contain a summary, description, return value, cross-reference ("See Also"), and example. The summary is a description of syntax, showing you the number and type of arguments to give when calling the function. The description explains the effects of the function and gives further information about arguments. The return value typically indicates whether or not the function was successful. In addition, some functions return a file handle or a length.

Finally, the cross-reference refers you to other functions that you may need to use in combination with the function described. For example, many functions cannot be performed unless you first call the **FileNameToHandle** function.

For a topical listing of these functions, see Table 8.2 in the previous chapter.

SUMMARY

#include <ext.h>

PFILE pascal AddFile (*p*)
char far **p*;

DESCRIPTION

The **AddFile** function opens a file for editing. The parameter *p* points to a null-terminated string of text containing the name of the file to open.

The file can be new or one that currently exists on disk; however, the file should not already be open for editing. Therefore, to open an existing file, first check to see if it is already open by calling **FileNameToHandle**.

After you open an existing file, you should immediately call the **FileRead** function to properly initialize the internal file buffer. If you open a new file, the file will not be added to the disk until you call **FileWrite**.

RETURN VALUE

The function returns a handle to the file.

SEE ALSO

FileNameToHandle, **FileRead**, **FileWrite**

EXAMPLE

The following example checks to see if the file MYDATA.FIL is currently open for editing; if not, **AddFile** is called to open the file. In either case, the file handle is assigned to pFile.

```
char *p = "MYDATA.FIL";

if( (pFile = FileNameToHandle( p, NULL )) == 0 )
{
        pFile = AddFile( p );
        FileRead( p, pFile );
}
```

SUMMARY

#include <ext.h>

flagType pascal BadArg (void)

DESCRIPTION

The **BadArg** function reports an error message stating that the user's argument is invalid. Usually you do not need to call this function because the editor looks at the type of your function as declared in **cmdTable** and rejects commands with the wrong argument type.

This function is primarily useful if your editing function does some additional tests for valid input, beyond argument type. For example, you may want to exclude each *numarg* larger than a certain value.

RETURN VALUE

The function does not return a value.

SEE ALSO

DoMessage

EXAMPLE

The following example causes the editor to report an invalid-argument message if the range of lines is greater than 10. Normally this condition is not an error, but you may want to write a function that restricts the size of an argument.

```
if( yEnd > yStart == 10 )
{
    BadArg( );
    return 0;
}
```

SUMMARY

#include <ext.h>

void pascal CopyBox (*pFileSrc, pFileDst, xLeft, yTop, xRight, yBottom, xDst, yDst*)
PFILE *pFileSrc, pFileDst*;
COL *xLeft, xRight, xDst*;
LINE *yTop, yBottom, yDst*;

DESCRIPTION

The **CopyBox** function copies the box delimited by the edges *xLeft*, *yTop*, *xRight*, and *yBottom* in the source file and inserts this box just before position (*xDst*, *yDst*) in the destination file. If the *pFileSrc* is null (0), the function inserts a box of blank spaces of the size implied by the coordinate parameters.

The parameters *pFileSrc* and *pFileDst* are handles to the source and destination files. The parameters *xLeft*, *xRight*, *yTop*, and *yBottom* specify the boundaries of the box, inclusive, of the text to be copied. The text is inserted into the destination file just before the location specified by *xDst, yDst*.

The same file can serve as source and destination. However, in that case the source and destination regions must not overlap.

All coordinates in lower-level functions are zero based.

RETURN VALUE

The function does not return a value.

SEE ALSO

CopyBox, CopyLine, CopyStream, FileNameToHandle

EXAMPLE

The following example copies a box from the file A.TXT and inserts this region into the file B.TXT:

```
pAFILE = FileNameToHandle( "A.TXT", NULL );
pBFILE = FileNameToHandle( "B.TXT", NULL );
.
.
.
CopyBox( pAFILE, pBFILE, Left, Top, Right, End, xBFILE, yBFILE );
```

SUMMARY

#include <ext.h>

void pascal CopyLine (*pFileSrc*, *pFileDst*, *yStart*, *yEnd*, *yDst*)
PFILE *pFileSrc*, *pFileDst*;
LINE *yStart*, *yEnd*, *yDst*;

DESCRIPTION

The **CopyLine** function can be used either to copy a group of lines from one region to another or to insert a blank line.

The *pFileSrc* and *pFileDst* parameters are handles to the source and destination files. If *pFileSrc* is null (0), the function inserts one or more blanks lines, the number of lines being determined by the relative values of *yStart* and *yEnd*. (The number of blank lines is equal to the difference between *yStart* and *yEnd* plus one.) Otherwise, the function copies the lines from *yStart* to *yEnd*, inclusive, in the destination file. Lines are inserted directly before line *yDst* in the destination file.

The **CopyLine** function should not be confused with the **PutLine** function. **PutLine** replaces a line and does not affect the total number of lines. **CopyLine** inserts one or more lines and therefore increases the length of the file.

The same file cannot serve as both source and destination. To copy text from one part of the file to another, copy to a temporary file, such as <clipboard>.

All line numbers in low-level functions are zero based.

RETURN VALUE

The function does not return a value.

SEE ALSO

CopyBox, **CopyStream**, **FileNameToHandle**, **PutLine**

EXAMPLE

The following code inserts a blank line at the beginning of the file:

```
cfile = FileNameToHandle( "", NULL );
CopyLines( NULL, cfile, 0, 0, 0 );
```

SUMMARY

#include <ext.h>

void pascal CopyStream (*pFileSrc*, *pFileDst*, *xStart*, *yStart*, *xEnd*, *yEnd*, *xDst*, *yDst*)
PFILE *pFileSrc*, *pFileDst*;
COL *xStart*, *xEnd*, *xDst*;
LINE *yStart*, *yEnd*, *yDst*;

DESCRIPTION

The **CopyStream** function copies the stream of text (including new lines) beginning at position (*xStart, yStart*), up to but not including position (*xEnd, yEnd*). The stream of text is inserted into the destination file just before position (*xDst, yDst*). If *pFileSrc* is null (0), a blank space is inserted.

The *pFileSrc* and *pFileDst* parameters are file handles to the source and destination files.

The same file cannot serve as both source and destination. To copy text from one part of the file to another, copy to a temporary file, such as <clipboard>.

RETURN VALUE

The function does not return a value.

SEE ALSO

CopyBox, **CopyLine**, **FileNameToHandle**

EXAMPLE

The following example copies a stream from the file A.TXT and inserts this region into the file B.TXT:

```
pAFILE = FileNameToHandle( "A.TXT", "" );
pBFILE = FileNameToHandle( "B.TXT", "" );
.
.
.
CopyStream( pAFILE, pBFILE, xStart, yStart, xEnd, yEnd, xBFILE,
yBFILE );
```

SUMMARY

#include <ext.h>

void pascal DelBox (*pFile*, *xLeft*, *yTop*, *xRight*, *yBottom*)
PFILE *pFile*;
COL *xLeft*, *xRight*;
LINE *yTop*, *yBottom*;

DESCRIPTION

The **DelBox** function deletes all spaces in the box delimited by the positions (*xLeft*, *yTop*) and (*xRight*, *yBottom*). The box includes all four edges in the parameter list. The *pFile* parameter is a handle to the file to be modified.

All line and column coordinates are zero based.

RETURN VALUE

The function does not return a value.

SEE ALSO

DelLine, **DelStream**, **FileNameToHandle**

EXAMPLE

The following example deletes the user-defined box argument:

```
pFile = FileNameToHandle( "", NULL );
.
.
.
Left   = pArg->arg.boxarg.xLeft;
Right  = pArg->arg.boxarg.xRight;
Top    = pArg->arg.boxarg.yTop;
Bottom = pArg->arg.boxarg.yBottom;
.
.
.
DelBox( pFile, Left, Top, Right, Bottom );
```

SUMMARY

#include <ext.h>

void pascal DelFile (*pFile*)
PFILE *pFile*;

DESCRIPTION

The *DelFile* function deletes the entire contents of an internal file buffer. The effect of deleting contents can be made permanent by calling the **FileWrite** function, which replaces the contents of the file on disk with the contents of the internal file buffer.

The parameter *pFile* is the handle of the file to be cleared.

RETURN VALUE

The function does not return a value.

SEE ALSO

AddFile, FileNameToHandle, FileWrite

EXAMPLE

The following example deletes the contents of the file JUNK.TXT, then calls **FileWrite** to make the change permanent:

```
if( (pFile = FileNameToHandle( "JUNK.TXT", NULL)) == 0 )
        pFile = AddFile(  "JUNK.TXT"  );

DelFile( pFile );
FileWrite( "JUNK.TXT", pFile );
```

SUMMARY

#include <ext.h>

void pascal DelLine (*pFile*, *yStart*, *yEnd*)
PFILE *pFile*;
LINE *yStart*, *yEnd*;

DESCRIPTION

The **DelLine** function deletes lines *yStart* through *yEnd*, inclusive, in the file *pFile*.

The *pFile* parameter is a handle to a file from which lines are to be deleted. *yStart* is the first line to be deleted, and *yEnd* is the last line to be deleted.

All line coordinates for low-level functions are zero based.

RETURN VALUE

The function does not return a value.

SEE ALSO

DelBox, **DelStream**, **FileNameToHandle**

EXAMPLE

The following example deletes the user-defined line argument:

```
cfile = FileNameToHandle( "", NULL );
.
.
.
Start  = pArg->arg.linearg.xStart;
End    = pArg->arg.linearg.xEnd;
.
.
.
DelLine( cfile, Start, End );
```

SUMMARY

#include <ext.h>

void pascal DelStream (*pFile*, *xStart*, *yStart*, *xEnd*, *yEnd*)
PFILE *pFile*;
COL xStart, x*End*
LINE *yStart*, *yEnd*;

DESCRIPTION

The **DelStream** function deletes a stream of text beginning with a starting coordinate up to but not including the ending coordinate.

The *xStart* and *yStart* parameters give the coordinates of the beginning of the stream. The *xEnd* and *yEnd* parameters give the coordinates of the byte just after the end of the stream.

All column and line coordinates for low-level functions are zero based.

RETURN VALUE

The function does not return a value.

SEE ALSO

DelBox, **DelLine**, **FileNameToHandle**

EXAMPLE

The following example deletes the user-defined stream of text:

```
cfile = FileNameToHandle( "", NULL );
.
.
.
xStart = pArg->arg.streamarg.xStart;
yStart = pArg->arg.streamarg.yStart;
xEnd   = pArg->arg.streamarg.xEnd;
yEnd   = pArg->arg.streamarg.yEnd;
.
.
.
DelStream( cfile, xStart, yStart, xEnd, yEnd );
```

SUMMARY

#include <ext.h>

void pascal Display ( )

DESCRIPTION

The **Display** function refreshes the screen by examining editing changes and making the minimum screen changes necessary. A keystroke interrupts the function and causes immediate return.

The editor normally updates the display whenever the editing session is "idle"; that is, when the editor is waiting for the next command from the user. Therefore, it is usually not necessary to call the **Display** function. However, if your function runs for an extended period of time, call **Display** periodically to show the user the stage of any intermediate changes. Otherwise, the results of these changes are not displayed until completion of the function.

RETURN VALUE

The function does not return a value.

EXAMPLE

```
Display();
```

SUMMARY

#include <ext.h>

int pascal DoMessage (*pStr*)
char far **pStr*;

DESCRIPTION

The **DoMessage** function writes a message to the dialog line. The *pStr* parameter points to a null-terminated string of text containing the message to be written.

RETURN VALUE

The function returns the number of characters written.

EXAMPLE

The following example outputs a message on the dialog line as part of the initialization procedure **WhenLoaded**:

```
WhenLoaded ()
{
        .
        .
        .

        DoMessage( "My extension now loaded" );
}
```

SUMMARY

#include <ext.h>

flagType pascal fExecute(*pStr*)
char far*pStr;

DESCRIPTION

The **fExecute** function executes a macro, using the standard rules for macro execution. The pointer *pStr* points to a null-terminated string of text containing the macro to be executed.

You may sometimes find it convenient to invoke a predefined editing function by calling **fExecute**. For example, to search for the next occurrence of a given string, you can either write a loop that examines each line in the file, or simply invoke *Psearch* by calling **fExecute**. Macros are especially convenient when you want to look for a regular expression.

RETURN VALUE

The function passes along the value TRUE (nonzero) or FALSE (zero) returned by the last function the macro executed.

EXAMPLE

The following example invokes a macro in order to search for the next occurrence of the regular expression `RegEx`:

```
strcpy( buf, "arg arg \"" );
strcat( buf, RegEx );
strcat( buf, "\" psearch" );

if( fExecute (buf) )
{
    GetCursor( xCur, yCur );       /* Get new coordinates  */
}
else
{
.
.
.
 /* Take action for item not found */
.
.
.
}
```

SUMMARY

#include <ext.h>

LINE pascal FileLength *(pFile)*
PFILE *pFile*;

DESCRIPTION

The **FileLength** function determines the length of the file pointed to by *pFile*.

This function is useful for global operations in which it is necessary to know when you have reached the end of the file.

RETURN VALUE

The function returns the number of lines in the given file.

SEE ALSO

FileNameToHandle

EXAMPLE

The following example determines the number of lines in the current file:

```
LINE fileLen;

cfile = FileNameToHandle( "", NJLL );
fileLen = FileLength( cfile );
```

SUMMARY

#include <ext.h>

PFILE pascal FileNameToHandle (*pname*, *pShortName*)
char **pname*, **pShortName*;

DESCRIPTION

The **FileNameToHandle** function returns the handle of a file already opened for editing.

The *pname* parameter points to a null-terminated string of text containing a complete file name. **FileNameToHandle** looks for an exact match in its list of open files. The full path name must match. If the string does not specify a path to a directory, the current directory is assumed.

If the function cannot find a match to *pname*, it attempts to match the *pShortName* parameter. This parameter points to a null-terminated string of text containing only a base file name—the function ignores any path name or extension in the short name. The editor selects the first file name with a base name matching the short name.

If the first string is empty, the function returns a handle to the current file. The editor does not try to match a short name if the second parameter is a null pointer or points to an empty string.

RETURN VALUE

The **FileNameToHandle** function returns the handle to the given file. If the given file is not open for editing, the function returns NULL.

EXAMPLE

The following example returns a handle to the current file:

```
PFILE  curfile;

curfile = FileNameToHandle( "", NULL );
```

SUMMARY

#include <ext.h>

flagType pascal FileRead *(name, pFile)*
char far *name;
PFILE *pFile;*

DESCRIPTION

The **FileRead** function reads the contents of the specified disk file and stores them in the internal file buffer specified by the *pFile*. The old contents of the file buffer are lost.

The parameter *name* is a pointer to a null-terminated string containing the name of the disk file to be read. The *pFile* parameter is the handle of the internal file buffer to write the data to.

When you open a file for editing with **AddFile**, the file buffer is initially empty. Call **FileRead** to initialize the buffer with the current contents of the file.

RETURN VALUE

The function returns TRUE (nonzero) if the copy is successful and FALSE (zero) if not.

SEE ALSO

AddFile, FileWrite, FileNameToHandle

EXAMPLE

The following example opens the file MYTEXT.FIL for editing and then initializes the buffer with the current contents of the file:

```
if( (pFile = FileNameToHandle("MYTEXT.FIL",NULL)) == 0 )
{
        pFile = AddFile( "MYTEXT.FIL" );
        FileRead( "MYTEXT.FIL", pFile );
}
```

SUMMARY

#include <ext.h>

flagType pascal FileWrite *(savename, pFile)*
char far *savename;
PFILE *pFile*;

DESCRIPTION

The **FileWrite** function writes the contents of the specified file buffer out to a disk file.

The *pFile* parameter is the handle to the file buffer. The *savename* parameter points to the name of the disk file. If *savename* points to an empty string, the function writes to the disk file with the name as the file *pFile*. (Note that a file handle points to an internal structure that contains the name of the file as well as other data.)

The function first writes contents to a temporary file. If the write operation is successful, the temporary file is renamed to the destination file.

You need not use **FileWrite** with the current file. Since the user is currently editing this file, you can let the user decide when to save the file to disk. However, **FileWrite** should be used with other files that you open for editing.

RETURN VALUE

The function returns TRUE (nonzero) if the copy was successful and FALSE (zero) if not.

SEE ALSO

AddFile, FileRead, FileNameToHandle

EXAMPLE

The following example alters the contents of the file JUNK.TXT, then makes the deletion permanent by calling **FileWrite**:

```
char *p = "JUNK.TXT";

if( (pFile = FileNameToHandle(p, NULL)) == 0 )
{
    pFile = AddFile( p );
    FileRead( p, pFile );
}
.
.
.
/* Manipulate data in the file */
.
.
.
FileWrite( p, pFile );
```

SUMMARY

#include <ext.h>

void pascal GetCursor *(px, py)*;
COL far **px*;
LINE far **py*;

DESCRIPTION

The **GetCursor** function indicates current cursor position by modifying the variables to which *px* and *py* point. The function sets **px* to the current cursor column, and **py* to the current cursor line.

Upon return, the numbers pointed to by *px* and *py* indicate the column and row, respectively, of the current cursor position.

RETURN VALUE

The function does not return a value.

SEE ALSO

MoveCur

EXAMPLE

```
LINE  yCur;
COL   xCur;
.
.
.
        GetCursor( &xCur, &yCur );
```

SUMMARY

#include <ext.h>

int pascal GetLine (*line*, *buf*, *pFile*)
LINE *line*;
char far **buf*
PFILE *pFile*;

DESCRIPTION

The **GetLine** function is the principal means for reading text from a file buffer.

The function reads a specified line of text and copies the line into a character string pointed to by *buf*. The editor terminates the string with a null value. The *line* parameter contains a line number to read. In extensions, line numbers are always zero based. The *pFile* parameter is the handle to the file.

If the **realtabs** switch is off, the function expands tabs to spaces (as indicated by the **entab** and **filetab** switches) before copying the text to *buf*.

RETURN VALUE

The function returns the number of characters in the line after any tab conversion.

SEE ALSO

CopyLine, **PutLine**, **FileNameToHandle**

EXAMPLE

The following example reads the line of text that includes the initial cursor position and copies it into *buf*:

```
PFILE cfile;
LINE  yCur;
COL   xCur;
 char  buf[BUFLEN];
int   len; .
.
.
    cfile = FileNameToHandle( "", NULL );
    GetCursor( &xCur, &yCur );
    len = GetLine( yCur, buf, cfile );
```

SUMMARY

#include <ext.h>

void pascal KbHook( );

DESCRIPTION

The **KbHook** function reverses the effect of the **KbUnHook** function and restores normal keyboard-input reading by the editor.

RETURN VALUE

The function does not return a value.

SEE ALSO

KhUnHook

EXAMPLE

```
KbHook( );
```

SUMMARY

#include <ext.h>

void pascal KbUnHook( );

DESCRIPTION

The **KbUnHook** function changes the focus of the keyboard so that keyboard input is no longer read by the editor. When attempting to use system-level calls to read from the keyboard, it is necessary to first call this function.

In particular, it is necessary to call the **KbUnHook** function before transferring control to a program that reads directly from the keyboard by using operating-system or BIOS calls, or by working directly with hardware.

You normally do not need to call this function. For most work with the keyboard, use the **ReadChar** function. **ReadChar** lets the editor read the keyboard for you, but allows you to intercept the keystroke and evaluate it in any way you choose. Use **KbUnHook** only for situations that **ReadChar** cannot accommodate.

RETURN VALUE

The function does not return a value.

SEE ALSO

KbHook

EXAMPLE

```
KbUnHook( );
```

SUMMARY

#include <ext.h>

void pascal MoveCur (*x*, *y*)
COL *x*;
LINE *y*;

DESCRIPTION

The **MoveCur** function moves the cursor to the specified position within the current file. If the cursor is within the same window, no window movement occurs. Otherwise, the window scrolls as needed, and the cursor is placed at a common position specified by the numeric switch **hike.**

After the function is called, the cursor moves to column *x*, line *y* of the current file. The editing window scrolls, if necessary, to display this position within the window.

RETURN VALUE

The function does not return a value.

SEE ALSO

GetCursor

EXAMPLE

The following example edits the file but restores the initial cursor position:

```
GetCursor( &xCur, &yCur );
.
.
.
/* Modify the file */
.
.
.
MoveCur( xCur, yCur );
```

SUMMARY

#include <ext.h>

void pascal pFileToTop *(pFileTmp)*
PFILE *pFileTmp*;

DESCRIPTION

The **pFileToTop** function selects a file as the current file and makes it visible in the current window. The function accomplishes this operation by moving the specified file handle to the top of the file list for the current window. (This list is stored in the <information-file> pseudo file.)

The parameter *pFileTmp* is the file handle to move to the top of the file list.

RETURN VALUE

The function does not return a value.

SEE ALSO

AddFile, **FileNameToHandle**

EXAMPLE

```
pFileToTop( pFile );
```

SUMMARY

#include <ext.h>

void pascal PutLine (*line*, *buf*, *pFile*)
LINE *line*;
char far **buf*;
PFILE *pFile*;

DESCRIPTION

The **PutLine** function is the principal means for writing text to a file buffer.

The function replaces a single line of text. The parameter *buf* points to the string that contains the new line of text. This string should terminate with a null value, but it should not contain a new-line character.

The parameter *line* contains the line number at which the replacement is to take place. Line numbers start at 0; if *line* has the value 0, the new line of text is inserted at the beginning of the file.

If *line* is greater than the number of lines in the file, **PutLine** inserts empty lines at the end of the file.

RETURN VALUE

The function does not return a value.

SEE ALSO

CopyLine, **GetLine**, **FileLength**, **FileNameToHandle**

EXAMPLE

The following code replaces the first line of the current file with the string pointed to by `buf`:

```
PutLine( 0, buf, cfile );
```

SUMMARY

#include <ext.h>

long pascal ReadChar( );

DESCRIPTION

The **ReadChar** function returns the next keystroke typed. The editor does not echo the keystroke or invoke a function. Once intercepted, the keystroke cannot be placed back into the keyboard buffer for execution. The function returns a long integer composed of four bytes containing information about the keystroke:

Byte	Description
0	ASCII character
1	scan-code character
2	shift information, in xxNxACxS format described below
3	null character (0), which is unused

Byte 2 provides information about the shift-key conditions (N)UMLOCK, (A)LT, (C)TRL, and (S)HIFT, in the format xxNxACxS. Each x indicates an unused bit. The bits N, A, C, and S are each on or off, depending on the associated condition. For example, if the ALT, CTRL, and SHIFT conditions are all on, but the NUMLOCK condition is off, byte 2 is returned as 00001101. Note: the N bit is 0 unless the key pressed is on the numeric keypad.

RETURN VALUE

The function returns a long integer containing the keystroke information.

SEE ALSO

ReadCmd, KbHook

EXAMPLE

```
#define ASCIIBYTE   0x000000FF
#define SCANBYTE    0x0000FF00
#define CTRLBIT     0x00040000
#define SHIFTBIT    0x00010000
#define NUMLKBIT    0x00200000

long    keystroke;
int     ascii_key, scan_byte;
int     contrl_on, shift_on, numlk_on;

        keystroke = ReadChar();
        ascii_key = keystroke & ASCIIBYTE;
        scan_byte = (keystroke & SCANBYTE) >> 8;
        contrl_on = (keystroke & CTRLBIT) > 0;
        shift_on  = (keystroke & SHIFTBIT) > 0;
        numlk_on  = (keystroke & NUMLKBIT) > 0;
```

SUMMARY

#include <ext.h>

PCMD pascal ReadCmd( );

DESCRIPTION

The **ReadCmd** function waits for input from the user. The next keystroke is translated into a function reference (according to current assignments), but the function is not executed. Instead, the editor passes information about the function in the form of a structure of type **cmdDesc**. Once intercepted, the keystroke cannot be placed back for execution.

RETURN VALUE

The return value is a structure of type **cmdDesc**. The structure describes the command corresponding to the keystroke pressed.

SEE ALSO

ReadChar

SUMMARY

#include <ext.h>

flagType pascal RemoveFile (*pFileRem*)
PFILE *pFileRem*;

DESCRIPTION

The **RemoveFile** function removes a file handle from memory, along with the file buffer and all other memory-resident information about the file. Calling this function helps to free up main memory, but it has no effect on the file as stored on disk.

This function is the converse of the **AddFile** function. **RemoveFile** closes a file. In other words, the file is no longer open for editing. However, unlike the C function **fclose**, the **RemoveFile** function does not force the writing of the file buffer to disk.

The parameter *pFileRem* is the file handle of the file to be removed.

RETURN VALUE

The function does not return a value.

SEE ALSO

AddFile, **FileNameToHandle**

EXAMPLE

```
RemoveFile( pNewFile );
```

SUMMARY

#include <ext.h>

flagType pascal Replace (*c*, *x*, *y*, *pFile*, *fInsert*)
char *c*;
COL *x*;
LINE *y*;
PFILE *pFile*;
flagType *fInsert*;

DESCRIPTION

The **Replace** function inserts or replaces characters one at a time. The *c* parameter contains the new character. The *x* and *y* parameters indicate the file position—by column and line—where the edit is to take place. Line numbers are zero based.

The *pFile* parameter is a file parameter handle returned by the **FileNameToHandle** function. To specify insertion, set *fInsert* to TRUE (nonzero). To specify replacement, set *fInsert* to FALSE (zero).

RETURN VALUE

The function returns TRUE (nonzero) if the edit is successful and FALSE (zero) otherwise.

SEE ALSO

PutLine

EXAMPLE

The following code inserts the word "Hello" at line `y` and column `x` of the current file:

```
#define  TRUE  -1
char     *p; PFILE   cfile;  /* handle to current file */
.
.
.
cfile = FileNameToHandle( "", NULL );  /* initialize cfile */
for ( p = "Hello"; *p; p++, y++ )
    Replace( *p, x, y, cfile, TRUE );
```

SUMMARY

#include <ext.h>

flagType pascal SetKey (*name*, *p*)
char far **name*, **far** **p*;

DESCRIPTION

The **SetKey** function assigns an editing function to a key.

The *name* parameter points to a string containing the name of the function, and the *p* parameter points to a string that names the key.

RETURN VALUE

The function returns TRUE (nonzero) if the assignment is successful and FALSE (zero) otherwise.

SEE ALSO

RemoveFile, **Replace**

EXAMPLE

The following code assigns the CTRL+X key to the newly defined function `NewFunc`:

```
SetKey( "NewFunc", "ctrl+x" );
```

Appendixes

Appendix A

Reference Tables

A.1 Categories of Editing Functions

Table A.1 lists the editing functions by category and gives a brief description of each function.

Table A.1 Summary of Editing Functions by Category

Command Manipulation	Description
Arg	Introduces an argument or function
Assign	Assigns function to a keystroke
Boxstream	Toggles between box and stream mode
Cancel	Cancels current operation
Execute	Executes an editor function or macro list
Graphic	Inserts the ASCII value of the key into the file
Lastselect	Recalls the last cursor-movement argument
Lasttext	Recalls the last *textarg* entered
Meta	Turns on the *Meta* command prefix
Quote	Treats next character literally
Repeat	Repeats the previous command
Undo	Reverses the effect of the last editing change
File Operations	**Description**
Exit	Exits the editor, with or without saving
Noedit	Toggles the no-edit restriction
Paste	Merges file or program output
Refresh	Rereads file, discarding edits
Saveall	Saves all modified files
Setfile	Saves current file or loads a new file
Cursor Movement	**Description**
Backtab	Moves cursor left to previous tab stop
Begfile	Moves cursor to beginning of file
Begline	Moves cursor left to beginning of line
Down	Moves cursor down one line

Table A.1 *(continued)*

Cursor Movement	Description
Endfile	Moves cursor to end of file
Endline	Moves cursor to right of last character of line
Home	Moves cursor to upper-left corner of window
Left	Moves cursor left one character
Mpage	Moves cursor back by one page
Mpara	Moves cursor back by paragraphs
Mword	Moves cursor back by words
Newline	Moves cursor down to next line
Ppage	Moves cursor forward by one page
Ppara	Moves cursor forward by paragraphs
Pword	Moves cursor forward by words
Right	Moves cursor right one character
Tab	Moves cursor right to next tab stop
Up	Moves cursor up one line
Mark/Goto Position	**Description**
Mark	Moves cursor to specified position in file
Restcur	Restores cursor position saved with *Savecur*
Savecur	Saves cursor position for use with *Restcur*
Windows	**Description**
Mlines	Moves window back by lines
Plines	Moves window forward by lines
Setwindow	Redisplays window
Window	Creates, removes, or moves between windows
Searching/Replacing	**Description**
Mgrep	Searches a series of files
Mreplace	Replaces throughout a series of files
Msearch	Searches backward
Psearch	Searches forward
Qreplace	Replaces with confirmation
Replace	Replaces without confirmation
Searchall	Highlights all occurrences of a string

Table A.1 *(continued)*

Special Insert	Description
Curdate	Inserts current date (e.g., 28-Nov-1988)
Curday	Inserts current day (Sun...Sat)
Curfile	Inserts name of current file
Curfileext	Inserts extension of current file
Curfilenam	Inserts base name of current file
Curtime	Inserts current time (e.g., 13:45:55)
Inserting/Deleting Text	**Description**
Cdelete	Deletes character to left, excluding line breaks
Copy	Copies lines to the Clipboard
Delete	Deletes the highlighted area
Emacscdel	Deletes character to left, including line breaks
Emacsnewl	Starts new line, breaking current line
Insert	Inserts spaces into the highlighted area
Ldelete	Deletes lines into the Clipboard
Linsert	Inserts blank lines
Paste	Inserts text from the Clipboard
Sdelete	Deletes stream of text, including line breaks
Sinsert	Inserts blanks, breaking lines if necessary
Programming	**Description**
Argcompile	Performs the *Arg Compile* command
Compile	Executes compile or build command
Nextmsg	Moves cursor to next error message
Pbal	Balances parentheses and brackets
Macro Creation	**Description**
Assign	Defines a macro
Message	Displays message on the dialog line
Record	Turns macro recording on or off
Tell	Displays assignment or macro definition

Table A.1 *(continued)*

Miscellaneous	Description
Environment	Executes and views environment settings
Information	Displays list of previously edited files
Initialize	Rereads initialization file
Insertmode	Toggles insert mode on and off
Print	Prints all or part of a file
Shell	Spawns a system-level shell or command line

A.2 Key Assignments for Editing Functions

Table A.2 lists the editing functions and the assigned keys for each of the configurations provided with the setup program.

Table A.2 Function Assignments

Function	Default	Quick/ WordStar®	BRIEF	Epsilon
Arg	ALT+A	ALT+A	ALT+A	CTRL+U or CTRL+X
Argcompile	---	F5	ALT+F10	F5
Assign	ALT+=	ALT+=	F7	F1
Backtab	SHIFT+TAB	SHIFT+TAB	SHIFT+TAB	SHIFT+TAB
Begfile	CTRL+PGUP	---	---	---
Begline	HOME	HOME or CTRL+QS	HOME	CTRL+A
Boxstream	CTRL+B	---	---	---
Cancel	ESC or CTRL+BREAK	ESC	ESC	CTRL+C
Cdelete	CTRL+G	CTRL+G	BKSP	---
Compile	CTRL+F3	SHIFT+F3	CTRL+N	SHIFT+F3
Copy	CTRL+INS or press + (keypad)	CTRL+INS	+ (keypad)	ALT+W
Curdate	---	---	---	---
Curday	---	---	---	---
Curfile	---	---	---	---
Curfileext	---	---	---	---
Curfilenam	---	---	---	---
Curtime	---	---	---	---
Delete	DEL	---	---	---
Down	DOWN or CTRL+X	DOWN or CTRL+X	DOWN	DOWN or CTRL+N
Emacscdel	BKSP	BKSP	---	BKSP or CTRL+H
Emacsnewl	ENTER	ENTER	---	ENTER
Endfile	CTRL+PGDN	---	---	---
Endline	END	END or CTRL+QD	END	CTRL+E
Environment	---	---	---	---

Table A.2 *(continued)*

Function	Default	Quick/ WordStar®	BRIEF	Epsilon
Execute	F7	F10	F10	ALT+X
Exit	F8	ALT+X	ALT+X	F8
Home	CTRL+HOME	CTRL+HOME	CTRL+HOME	HOME
Information	SHIFT+F10	SHIFT+F1	ALT+B	SHIFT+F1
Initialize	SHIFT+F8	ALT+F10	SHIFT+F10	ALT+F10
Insert	---	---	---	---
Insertmode	INS or CTRL+V	INS or CTRL+V	ALT+I	CTRL+V
Lastselect	CTRL+U	---	---	---
Lasttext	CTRL+O	ALT+L	ALT+L	ALT+L
Ldelete	CTRL+Y	CTRL+Y	ALT+D	CTRL+K
Left	LEFT or CTRL+S	LEFT	LEFT	LEFT or CTRL+B
Linsert	CTRL+N	CTRL+N	CTRL+ENTER	CTRL+O
Mark	CTRL+M	ALT+M	ALT+M	CTRL+@
Message	---	---	---	---
Meta	F9	F9	F9	F9
Mgrep	---	---	---	---
Mlines	CTRL+W	CTRL+W	ALT+U	CTRL+W
Mpage	PGUP or CTRL+R	PGUP or CTRL+R	PGUP	PGUP or ALT+V
Mpara	---	CTRL+PGUP	CTRL+PGUP	ALT+UP
Mreplace	---	---	---	---
Msearch	F4	F4	ALT+F5	CTRL+R
Mword	CTRL+LEFT or CTRL+A	CTRL+LEFT	CTRL+LEFT	CTRL+LEFT or ALT+B
Newline	---	---	ENTER	---
Nextmsg	SHIFT+F3	---	---	---
Noedit	---	---	---	---
Paste	SHIFT+INS	SHIFT+INS	INS	CTRL+Y or INS
Pbal	CTRL+[	CTRL+[	CTRL+[	CTRL+[
Plines	CTRL+Z	CTRL+Z	CTRL+Z	CTRL+Z
Ppage	PGDN or CTRL+C	PGDN or CTRL+C	PDGN	PDGN
Ppara	---	CTRL+PGDN	CTRL+PDGN	ALT+DOWN
Print	CTRL+F8 or ALT+F2	---	---	---

Table A.2 *(continued)*

Function	Default	Quick/ WordStar®	BRIEF	Epsilon
Psearch	F3	F3	F5	F4 or CTRL+S
Pword	CTRL+RIGHT or CTRL+F	CTRL+RIGHT or CTRL+F	CTRL+RIGHT	CTRL+RIGHT or ALT+F
Qreplace	CTRL+\	ALT+F3	F6	ALT+F3 or ALT+5 or ALT+8
Quote	CTRL+P	ALT+Q	ALT+Q	CTRL+Q
Record	ALT+R	---	---	---
Refresh	SHIFT+F7	ALT+R	CTRL+]	ALT+R
Repeat	---	---	---	---
Replace	CTRL+L	CTRL+L	SHIFT+F6	---
Restcur	---	---	---	---
Right	RIGHT or CTRL+D	RIGHT or CTRL+D	RIGHT	RIGHT or CTRL+F
Saveall	---	---	---	---
Savecur	---	---	---	---
Sdelete	---	DEL	DEL or press – (keypad)	DEL or CTRL+D
Searchall	SHIFT+F6	---	---	---
Setfile	F2	F2	ALT+N	F2
Setwindow	CTRL+]	CTRL+]	F2	CTRL+]
Shell	SHIFT+F9	SHIFT+F9	ALT+Z	ALT+Z
Sinsert	CTRL+J	ALT+INS	CTRL+INS	ALT+INS
Tab	TAB	TAB	TAB	TAB or CTRL+I
Tell	CTRL+T	---	---	---
Undo	ALT+BKSP	ALT+BKSP	* (keypad)	CTRL+BKSP
Up	UP or CTRL+E	UP or CTRL+E	UP	UP or CTRL+P
Window	F6	F6	F1	ALT+PGDN

A.3 Comprehensive Listing of Editing Functions

Table A.3 gives a comprehensive listing of the editing functions and syntax for each command. Default keystrokes, if available, are given in parentheses.

Table A.3 Comprehensive List of Functions

Function (and Default Keystrokes)	Syntax	Description
Arg (ALT+A)	*Arg*	Introduces a function or an argument for a function.
Argcompile	*Argcompile*	Performs the *Arg Compile* command. A macro for this function appears in the TOOLS.PRE file.
Assign (ALT+=)	*Assign*	Treats the entire line (except for the line break) on which the cursor is positioned as a function assignment or macro definition.
	Arg Assign	Treats the text from the initial cursor position to the end of the line (not including the line break) as a function assignment or macro definition.
	Arg boxarg Assign *Arg markarg Assign* *Arg numarg Assign*	Treats each line of the *boxarg* as an individual function assignment or macro definition.
	Arg linearg Assign	Treats each line as a separate function assignment or macro definition, ignoring blank lines.
	Arg textarg Assign	Treats *textarg* as a function assignment or macro definition.
	Arg ? *Assign*	Displays the current function assignments for all functions and macros.
Backtab (SHIFT+TAB)	*Backtab*	Moves the cursor to the previous tab stop. Tab stops are defined to be every *n*th character, where *n* is defined by the **tabstops** switch.
Begfile (CTRL+PGUP)	*Begfile*	Places the cursor at the beginning of the file.
Begline (HOME)	*Begline*	Places the cursor on the first nonblank character on the line.
	Meta Begline	Places the cursor in the first character position of the line.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Boxstream (CTRL+B)	*Boxstream*	Toggles between box mode and stream mode. In box mode, each cursor-movement is interpreted as a rectangular-shaped *linearg* or *boxarg*. In stream mode, the editor highlights all file positions between initial and new cursor position.
Cancel (ESC)	*Cancel*	Cancels the current operation in progress.
Cdelete (CTRL+G)	*Cdelete*	Deletes the previous character, excluding line breaks. If the cursor is in column 1, *Cdelete* moves the cursor to the end of the previous line. If issued in insert mode, *Cdelete* deletes the previous character, reducing the length of the line by 1; otherwise, it deletes the previous character and replaces it with a blank. If the cursor is beyond the end of the line when the function is invoked, the cursor is moved to the immediate right of the last character on the line.
Compile (CTRL+F3)	*Compile*	Displays status of the current compilation (if any) on the dialog line.
	Arg Compile	Compiles the current file. Uses the **extmake** command line that matches the filename extension of the current file.
	Arg textarg Compile	Uses the command line specified by **extmake:text**. The *textarg* replaces **%s** in the command line. See Table A.5 for more information on **extmake**.
	Arg Arg textarg Compile	Invokes the specified text as a program. The program is assumed to display its errors in the following format: *file row column message*.
	Arg Meta Compile	OS/2 only. Kills a protected-mode compilation running in the background, after prompting for confirmation.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Copy (CTRL+INS, or press + on numeric keypad)	*Copy*	Copies the current line into the Clipboard.
	Arg Copy	Copies text from the initial cursor position to the end of the line and places it into the Clipboard. Note that the line break is not picked up.
	Arg boxarg Copy *Arg linearg Copy* *Arg streamarg Copy* *Arg textarg Copy*	Copies the highlighted text into the Clipboard.
	Arg markarg Copy	Copies the range of text between the cursor and the location of the file marker into the Clipboard. In *stream* mode, a stream of text is selected. In *box* mode, the text is treated as a *boxarg* or *linearg* depending on the relative positions of the initial cursor position and the file marker.
	Arg numarg Copy	Copies the specified number of lines into the Clipboard, starting with the current line.
Curdate	*Curdate*	Inserts the current date at the cursor in the format of 28-Nov-1988.
Curday	*Curday*	Inserts the current day at the cursor in the format of Sun...Sat.
Curfile	*Curfile*	Inserts the fully qualified path name of the current file at the cursor.
Curfileext	*Curfileext*	Inserts the extension of the current file at the cursor.
Curfilenam	*Curfilenam*	Inserts the base name of the current file at the cursor.
Curtime	*Curtime*	Inserts the current time at the cursor in the format of 13:45:55.
Delete (DEL)	*Delete*	Deletes the single character under the cursor, excluding line breaks. The deleted character is not placed into the Clipboard.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
	Arg Delete	Deletes all text from the current cursor position to the end of the line. The deleted text (including the line break) is placed into the Clipboard. This command has the effect of joining lines.
	Arg boxarg Delete *Arg linearg Delete* *Arg streamarg Delete*	Deletes the highlighted text. The deleted text is placed into the Clipboard.
	Arg Meta Delete *Arg boxarg Meta Delete* *Arg linearg Meta Delete* *Arg streamarg Meta Delete*	Performs the deletions as described above, except the deleted text is not placed into the Clipboard.
Down (DOWN or CTRL+X)	*Down*	Moves the cursor down one line. If this would result in the cursor moving out of the window, the window is adjusted downward by the number of lines specified by the **vscroll** switch or less if in a small window.
	Meta Down	Moves the cursor to the bottom of the window without changing the column position.
Emacscdel (BKSP)	*Emacscdel*	Performs similarly to *Cdelete*, except that at the beginning of a line while in insert mode, *Emacscdel* deletes the line break between the current line and the previous line, joining the two lines together.
Emacsnewl (ENTER)	*Emacsnewl*	Performs similarly to *Newline*, except that when in insert mode, it breaks the current line at the cursor position.
Endfile (CTRL+PGDN)	*Endfile*	Places the cursor at the end of the file.
Endline (END)	*Endline*	Moves the cursor to the immediate right of the last nonblank character on the line.
	Meta Endline	Moves the cursor one character beyond the column corresponding to the rightmost edge of the window.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Environment	*Environment*	Executes the current line as an environment-variable setting. For example, assume the current line contains the following text: `PATH=C:\BIN;C:\DOS` The editor responds by adding this setting to the operating system environment space. This function is essentially the same as the system-level SET command. The editor recognizes the setting during the rest of the editing session, but the setting is lost when you exit the editor.
	Arg boxarg Environment *Arg linearg Environment*	Executes each highlighted line or line fragment as an environment-variable setting.
	Arg textarg Environment	Executes the text argument as an environment-variable setting.
	Arg **?** *Environment*	Displays all current environment-variable settings.
	Meta Environment	Performs environment "mappings" for all environment variables found on the current line, whenever the variable appears in the following syntax: **$(***environment-variable***)** For each such environment variable appearing on the line, the editor replaces the variable with the corresponding setting. For example, if PATH is set to `C:\BIN`, the editor replaces each occurrence of the text `$(PATH)` with the text `C:\BIN`.
	Arg Meta Environment	Performs environment mappings (see description above) for all text from the cursor position to the end of line.
	Arg linearg Meta Environment *Arg boxarg Meta Environment* *Arg streamarg Meta Environment*	Performs environment mappings (see description above) for all highlighted text.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Execute (F7)	*Arg Execute*	Treats the line from the initial cursor position to the end as a series of Microsoft Editor commands and executes them.
	Arg linearg Execute *Arg textarg Execute*	Treats the specified text as Microsoft Editor commands and executes them, following the standard rules of macro execution.
Exit (F8)	*Exit*	Saves the current file. If multiple files were specified on the command line, the editor advances to the next file. Otherwise, the editor quits and returns control to the operating system.
	Meta Exit	Performs similarly to *Exit*, except that the current file is not saved.
	Arg Exit	Performs similarly to *Exit*, except that if multiple files are specified on the command line, the editor exits without advancing to the next file.
	Arg Meta Exit	Performs similarly to *Arg Exit*, except that the editor does not save the current file.
Home (CTRL+HOME)	*Home*	Places the cursor in the upper-left corner of the current window.
Information (F10)	*Information*	Saves the current file and loads an information file that contains a list of all files in memory along with the current set of files that you have edited. The size of this list is controlled by the **tmpsav** switch, which has a default value of 20.
Initialize (SHIFT+F8)	*Initialize*	Reads all the editor statements from the [M] section of TOOLS.INI.
	Arg Initialize	Reads the editor statements from the TOOLS.INI file, using the continuous string of nonblank characters, starting with the initial cursor position, as the tag name.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
	Arg textarg Initialize	Reads all the editor statements from the [M-*textarg*] section of TOOLS.INI.
Insert	*Insert*	Inserts a single blank space at the current cursor position.
	Arg Insert	Inserts a carriage return at the initial cursor position, splitting the line.
	Arg streamarg Insert *Arg linearg Insert* *Arg boxarg Insert*	Inserts blank spaces into the highlighted area.
Insertmode (INS or CTRL+V)	*Insertmode*	Toggles between insert mode and overtype mode. If insert mode is on, `insert` appears on the status line. While in insert mode, each character that is entered is inserted at the cursor position, shifting the remainder of the line one position to the right. Overtype mode replaces the character at the cursor position with the character you type.
Lastselect (CTRL+U)	*Lastselect*	Recalls the last cursor-movement argument. This function produces the same result as returning to the last *Arg* position, invoking the *Arg* function, and then recreating the last cursor-movement argument.
Lasttext (CTRL+O)	*Lasttext*	Recalls the last *textarg*. This function produces the same result as invoking the *Arg* function and then retyping the previous *textarg*.
Ldelete (CTRL+Y)	*Ldelete*	Deletes the current line and places it into the Clipboard.
	Arg Ldelete	Deletes text, starting with the initial cursor position through the end of the line, and places it into the Clipboard. Note that it does not join the current line with the next line.
	Arg boxarg Ldelete *Arg linearg Ldelete*	Deletes the specified text from the file and places it into the Clipboard, treating the argument as a *linearg* or *boxarg* regardless of what mode the editor is in.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Left (LEFT or CTRL+S)	*Left*	Moves the cursor one character to the left. If this would result in the cursor moving out of the window, the window is adjusted to the left by the number of columns specified by the **hscroll** switch or less if in a small window.
	Meta Left	Moves the cursor to the left-most position in the window on the same line.
Linsert (CTRL+N)	*Linsert*	Inserts one blank line above the current line.
	Arg Linsert	Inserts or deletes blanks at the beginning of a line to make the first nonblank character appear under the cursor.
	Arg boxarg Linsert *Arg linearg Linsert*	Fills the specified area with blanks, treating the argument as a *linearg* or *boxarg* regardless of the editor's mode.
Mark (CTRL+M)	*Mark*	Moves the window to the beginning of the file.
	Arg Mark	Restores the window to its previous location. The editor remembers only the location prior to the last scrolling operation.
	Arg numarg Mark	Moves the cursor to the beginning of the line, where *numarg* specifies the position of the line in the file.
	Arg textarg Mark	Moves the cursor to the specified file marker. If the file marker was not previously defined, the editor uses the **markfile** switch to find the file that contains file marker definitions.
	Arg Arg textarg Mark	Deletes a marker definition.
	Arg Arg textarg Meta Mark	Defines a file marker at the initial cursor position. This does not record the file marker in the file specified by the **markfile** switch, but allows you to refer to this position as *textarg*.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Message	*Message*	Clears the dialog line.
	Arg textarg Message	Prints the text argument on the dialog line.
Meta (F9)	*Meta*	Modifies the action of the function it is used with. Refer to the individual functions for specific information.
Mgrep	*Mgrep*	Searches for the previously defined string or pattern. The editor searches all files listed in the `mgreplist` macro, which can contain DOS wildcards and environment variables, as in the following example: `megreplist:="DATA.FIL \ *.FOR $INCLUDE:*.H"` The editor places all strings found in the <compile> pseudo file.
	Arg Mgrep	Searches files for the string defined as the characters from the initial cursor position to the first blank character.
	Arg textarg Mgrep	Searches files for the specified text.
	Arg Arg Mgrep	Searches files for the regular expression defined as the characters from the initial cursor position to the first blank character.
	Arg Arg textarg Mgrep	Searches files for a regular expression as defined by *textarg*.
	Meta Mgrep *Arg Meta Mgrep* *Arg textarg Meta Mgrep* *Arg Arg Meta Mgrep* *Arg Arg textarg Meta Mgrep*	Performs similarly to command form above, except that value of the **case** switch is temporarily reversed.
Mlines (CTRL+W)	*Mlines*	Moves the window back by the number of lines specified by the **vscroll** switch or less if in a small window.
	Arg Mlines	Moves the window until the line that the cursor is on is at the bottom of the window.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
	Arg numarg Mlines	Moves the window back by the specified number of lines.
Mpage (PGUP or CTRL+R)	*Mpage*	Moves the window backward in the file by one window's worth of lines.
Mpara	*Mpara*	Moves the cursor to the first blank line preceding the current paragraph, or if currently on a blank line, the cursor is positioned before the previous paragraph.
	Meta Mpara	Moves the cursor to the first previous line that has text.
Mreplace	*Mreplace*	Performs a simple search-and-replace operation, prompting you for the search and replacement strings, and prompting at each occurrence for confirmation. The function searches all the file listed in the `mgreplist` macro, which can contain DOS wildcards and environment variables, as in the following example: `megreplist:="DATA.FIL \ *.FOR $INCLUDE:*.H"`
	Arg Arg Mreplace	Performs the same action as *Mreplace*, but uses regular-expression syntax.
Msearch (F4)	*Msearch*	Searches backward for the previously defined string or pattern. If the string or pattern is found, the window is moved to display it and the matched string or pattern is highlighted. If no match is found, no cursor movement takes place and a message is displayed.
	Arg Msearch	Searches backward in the file for the string defined as the characters from the initial cursor position to the first blank character.
	Arg textarg Msearch	Searches backward for the specified text.
	Arg Arg Msearch	Searches backward in the file for the regular expression defined as the characters from the initial cursor position to the first blank character.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
	Arg Arg textarg Msearch	Searches backward for a regular expression as defined by *textarg*.
	Meta Msearch *Arg Meta Msearch* *Arg textarg Meta Msearch* *Arg Arg Meta Msearch* *Arg Arg textarg Meta Msearch*	Performs similarly to command form above, except that value of the **case** switch is temporarily reversed.
Mword (CTRL+LEFT or CTRL+A)	*Mword*	Moves the cursor to the beginning of a word. If not in a word or at the first character, uses the previous word; otherwise, uses the current word.
	Meta Mword	Moves the cursor to the immediate right of the previous word.
Newline	*Newline*	Moves the cursor to a new line. If the **softcr** switch is set, the editor tries to place the cursor in an appropriate position based on the type of file. If the file is a C program, the editor tries to tab in based on continuation of lines and on open blocks. If the next line is blank, the editor places the cursor in the column corresponding to the first nonblank character of the previous line. If neither of the above is true, the editor places the cursor on the first nonblank character of the line.
	Meta Newline	Moves the cursor to column 1 of the next line.
Nextmsg (SHIFT+F3)	*Nextmsg*	Advances to next error message.
	Arg numarg Nextmsg	Moves forward or backward *numarg* error messages. A *numarg* value of 1 moves to next message; a value of -1 moves to previous message.
	Arg Nextmsg	Moves to the next error message (within current set of messages) that does not refer to current file.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
	Arg Arg Nextmsg	Positions the text-file cursor at the line with the error described in the error message at the current cursor position in the <compile> pseudo file. This message becomes the current error message. The following *Nextmsg* command displays the next error message from the <compile> pseudo file.
	Meta Nextmsg	OS/2 only. Advance to next "set" of error messages, in which a set corresponds to all the error messages for a single compilation. After executing this command, the previous set is deleted (though you can still view all subsequent sets of error messages in the <compile> pseudo file, until deleted).
Noedit	*Noedit*	Reverses the no-edit condition, so that if the editor was started with the /R (read only) option, this command removes the no-edit limitation. If the editor is not in the no-edit state, this command disallows all editing commands that alter a file.
	Meta Noedit	Reverses the no-edit condition for the current file.
Paste (SHIFT+INS)	*Paste*	Inserts the contents of the Clipboard prior to the current line if the contents were placed there in a line-oriented way, such as with *linearg* or *numarg*. Otherwise, the contents of the Clipboard are inserted at the current cursor position.
	Arg Paste	Inserts the text from the initial cursor position to the end of the line at the initial cursor position.
	Arg textarg Paste	Places the specified text into the Clipboard and inserts that text at the initial cursor position.
	Arg Arg textarg Paste	Interprets *textarg* as a file name and inserts the contents of that file into the current file above the current line.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
	Arg Arg !textarg Paste	Treats the text as a DOS command and inserts its output to **stdout** into the current file at the initial cursor position. The exclamation mark must be entered as shown.
Pbal (CTRL+[)	*Pbal*	Scans backward through the file, balancing parentheses and brackets. The first unbalanced one is highlighted when found. If it is found and is not visible, the editor displays the matching line on the dialog line, with the highlighted matching character. The corresponding character is placed into the file at the current cursor position. Note that the search does not include the current cursor position and that the scan only looks for more left brackets or parentheses than right, not just an unequal amount.
	Arg Pbal	Performs similarly to *Pbal* except that it scans forward in the file and looks for more right brackets or parentheses than left.
	Meta Pbal	Performs similarly to *Pbal* except that the file is not updated.
	Arg Meta Pbal	Performs similarly to *Arg Pbal* except that the file is not updated.
Plines (CTRL+Z)	*Plines*	Adjusts the window forward by the number of lines specified by the **vscroll** switch or less if in a small window.
	Arg Plines	Moves the window downward so the line that the cursor is on is at the top of the window.
	Arg numarg Plines	Moves the window forward the specified number of lines.
Ppage (PGDN or CTRL+C)	*Ppage*	Moves the window forward in the file by one window's worth of lines.
Ppara	*Ppara*	Moves the cursor forward one paragraph and places the cursor on the first line of the new paragraph.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
	Meta Ppara	Moves the cursor to the first blank line following the current paragraph.
Print (CTRL+F8)	*Print*	Prints the current file. If the **printcmd** switch is set, this function uses the system-level command given in the switch. Otherwise, the function copies output to LPT1.
	Arg textarg Print	Prints all the files listed in the text argument. File names should be separated with a space.
	Arg linearg Print *Arg boxarg Print* *Arg streamarg Print*	Prints the highlighted area.
Psearch (F3)	*Psearch*	Searches forward for the previously defined string or pattern. If the string or pattern is found, the window is moved to display it and the matched string or pattern is highlighted. If it is not found, no cursor movement takes place and a message is displayed.
	Arg Psearch	Searches forward in the file for the string defined as the characters from the initial cursor position to the first blank character.
	Arg textarg Psearch	Searches forward for the specified text.
	Arg Arg Psearch	Searches forward in the file for the regular expression defined as the characters from the initial cursor position to the first blank character.
	Arg Arg textarg Psearch	Searches forward for a regular expression as defined by *textarg*.
	Meta Psearch *Arg Meta Psearch* *Arg textarg Meta Psearch* *Arg Arg Meta Psearch* *Arg Arg textarg Meta Psearch*	Performs similarly to command form above, except that value of the **case** switch is temporarily reversed.
Pword (CTRL+RIGHT or CTRL+F)	*Pword*	Moves the cursor forward one word and places the cursor on the beginning of the new word.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
	Meta Pword	Moves cursor to immediate right of current word or, if not in a word, to the right of the next word.
Qreplace (CTRL+\)	*Qreplace*	Performs a simple search-and-replace operation, prompting you for the search and replacement strings, and prompting at each occurrence for confirmation. The search begins at the cursor position and continues through the end of the file.
	Arg boxarg Qreplace *Arg linearg Qreplace* *Arg streamarg Qreplace*	Perform the search-and-replace operation over the highlighted area, prompting at each occurrence for confirmation.
	Arg markarg Qreplace	Performs the search-and-replace operation between the initial cursor position and the specified file marker, prompting at each occurrence for confirmation.
	Arg numarg Qreplace	Performs the search-and-replace operation over the specified number of lines, starting with the current line, prompting at each occurrence for confirmation.
	Arg Arg Qreplace *Arg Arg boxarg Qreplace* *Arg Arg linearg Qreplace* *Arg Arg streamarg Qreplace* *Arg Arg markarg Qreplace* *Arg Arg numarg Qreplace*	Performs the same as the corresponding command listed above, except that the search pattern is a regular expression and the replacement pattern can select special tagged sections of the search for selective replacement. See Chapter 5 for more information.
Quote (CTRL+P)	*Quote*	Reads one keystroke from the keyboard and treats it literally. This is useful for inserting text into a file that happens to be assigned to an editor function.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Record (ALT+R)	*Record*	Turns on macro recording if off, and off if on. When a recording is stopped, the editor assigns all the recorded commands to the default macro name **recordvalue**. During the recording, the name of each command is written to the <record> pseudo file, which can be placed in a window and viewed as it is dynamically updated.
	Arg textarg Record	Turns on macro recording if off and gives recording the name specified in the text argument or turns recording off if on.
	Meta Record	Turns recording on if recording state is off, but no editing commands are executed until recording is turned off. Turns recording off if it is on.
	Arg Arg Record *Arg Arg textarg Record* *Arg Arg Meta Record*	Performs identically to the corresponding command listed above, but if the target macro already exists, editing commands are appended to the end of the macro.
Refresh (SHIFT+F7)	*Refresh*	Asks for confirmation and then rereads the file from disk, discarding all edits since the file was last saved.
	Arg Refresh	Asks for confirmation and then discards the file from memory, loading the last file edited in its place.
Repeat	*Repeat*	Repeats the last editing command, using precisely the same arguments and *Meta* condition used by the last command. However, the command is executed relative to the new cursor position. (Note: if the previous command had a cursor-movement argument, the text actually highlighted is reused as the argument.)

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
Replace (CTRL+L)	*Replace*	Performs a simple search-and-replace operation without confirmation, prompting you for the search string and replacement string. The search begins at the cursor position and continues through the end of the file.
	Arg boxarg Replace *Arg linearg Replace* *Arg streamarg Replace*	Performs the search-and-replace operation over the highlighted area.
	Arg markarg Replace	Performs the search-and-replace operation between the cursor and the specified file marker.
	Arg numarg Replace	Performs the search-and-replace operation over the specified number of lines, starting with the current line.
	Arg Arg Replace *Arg Arg boxarg Replace* *Arg Arg linearg Replace* *Arg Arg streamarg Replace* *Arg Arg markarg Replace* *Arg Arg numarg Replace*	Performs the same as the corresponding command listed above except that the search pattern is a regular expression and the replacement pattern can select special tagged sections of the search for selective replacement. See Chapter 5 for more information.
Restcur	*Restcur*	Restores the cursor position saved with *Savecur*.
Right (RIGHT or CTRL+D)	*Right*	Moves the cursor one character to the right. If this would result in the cursor moving out of the window, the window is adjusted to the right the number of columns specified by the **hscroll** switch or less if in a small window.
	Meta Right	Moves the cursor to the right-most position in the window.
Saveall	*Saveall*	Saves all files that have been altered during the current editing session without being saved.
Savecur	*Savecur*	Saves the current cursor position to be restored with *Restcur*.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Sdelete (DEL)	*Sdelete*	Deletes the single character under the cursor, excluding line breaks. It does not place the deleted character into the Clipboard. This command has the effect of joining lines.
	Arg Sdelete	Deletes from the current line at the point of the cursor position. The text deleted (including the line break) is placed into the Clipboard.
	Arg streamarg Sdelete	Deletes the stream of text from the initial cursor position up to the current cursor position and places it into the Clipboard.
Searchall (SHIFT+F6)	*Searchall*	Highlights all occurrences of the previously defined string or pattern. If at least one occurrence is found, the cursor moves to the first occurrence in the file.
	Arg Searchall	Highlights all occurrences of the string defined as the characters from the initial cursor position to the first blank character.
	Arg textarg Searchall	Highlights all occurrences of the specified text.
	Arg Arg Searchall	Highlights all occurrences of the regular expression defined as the characters from the initial cursor position to the first blank character.
	Arg Arg textarg Searchall	Highlights all occurrences of a regular expression as defined by *textarg*.
	Meta Searchall *Arg Meta Searchall* *Arg textarg Meta Searchall* *Arg Arg Meta Searchall* *Arg Arg textarg Meta Searchall*	Performs similarly to command forms above, except that the value of the **case** switch is temporarily reversed.
Setfile (F2)	*Setfile*	Switches to the most recently edited file, saving any changes made to the current file to disk.
	Arg Setfile	Switches to the file name that begins at the initial cursor position and ends with the first blank.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
	Arg textarg Setfile	Switches to the file specified by *textarg*. The text argument may be a drive or directory, in which case the editor changes the current drive or directory.
	Meta Setfile *Arg Meta Setfile* *Arg textarg Meta Setfile*	Performs similarly to the corresponding command listed above, but disables the saving of changes for the current file.
	Arg Arg textarg Setfile	Saves the current file under the name specified by *textarg*.
	Arg Arg Setfile	Saves the current file.
Setwindow (CTRL+])	*Setwindow*	Redisplays the entire screen.
	Arg Setwindow	Adjusts the window so that the initial cursor position becomes the home position (upper-left corner).
	Meta Setwindow	Redisplays the current line.
Shell (SHIFT+F9)	*Shell*	Saves the current file and runs the command shell.
	Meta Shell	Runs the command shell without saving the current file.
	Arg Shell	Uses the text on the screen from the cursor up to the end of line as a command to the shell.
	Arg boxarg Shell *Arg linearg Shell*	Treats each line of either argument as a separate command to the shell
	Arg textarg Shell	Uses *textarg* as a command to the shell.
Sinsert (CTRL+J)	*Sinsert*	Inserts a single blank space at the current cursor position.
	Arg Sinsert	Inserts a carriage return at the initial cursor position, splitting the line.
	Arg streamarg Sinsert	Insert a stream of blanks between the initial cursor position and the current cursor position.
Tab (TAB)	*Tab*	Moves the cursor to the next tab stop. Tab stops are defined to be every *n*th character, where *n* is defined by the **tabstops** switch.

Table A.3 (*continued*)

Function (and Default Keystrokes)	Syntax	Description
Tell (CTRL+T)	*Tell*	Prompts for a keystroke, then displays the name of the keystroke and the function assigned to it in the format *function:keyname*.
	Arg Tell	Identical to *Tell*, but if the key has a macro attached, displays *MacroName:=MacroValue*.
	Arg Arg Tell	Prompts for a keystroke, then displays the value of the macro attached to the key. If a function is assigned to the key, the editor displays the name of the function.
	Arg textarg Tell	Like *Arg Tell* but obtains the macro name from a *textarg* rather than a keystroke.
	Meta Tell *Arg Meta Tell* *Arg Arg Meta Tell* *Arg textarg Meta Tell*	Performs the same as the command listed above, except the editor inserts the output into the file rather than on the dialog line.
Undo (ALT+BKSP)	*Undo*	Reverses the last editing change. The maximum number of times this can be performed is set by the **undocount** switch.
	Meta Undo	Recalls a command previously canceled with *Undo*. This command is often called "redo."
Up (UP or CTRL+E)	*Up*	Moves the cursor up one line. If this would result in the cursor moving out of the window, the window is adjusted upward by the number of lines specified by the **vscroll** switch or fewer if in a small window.
	Meta Up	Moves the cursor to the top of the window without changing the column position.
Window (F6)	*Window*	Moves the cursor to the next window. With multiple windows, the next window is defined as being to the right of or below the current window.

Table A.3 *(continued)*

Function (and Default Keystrokes)	Syntax	Description
	Arg Window	Splits the current window horizontally at the initial cursor position. Note that all windows must be at least five lines high.
	Arg Arg Window	Splits the current window vertically at the initial cursor position. Note that all windows must be at least 10 columns wide.
	Meta Window	Closes the window.

A.4 Return Values of Editing Functions

Table A.4 gives an alphabetical listing of editing functions along with the conditions under which each function returns TRUE or FALSE. These return values are useful in conditional macros.

Table A.4 Editor Functions and Return Values

Function	Returns TRUE	Returns FALSE
Arg	Always	Never
Argcompile	Compile successful	Bad argument/compiler not found
Assign	Assignment successful	Invalid assignment
Backtab	Cursor moved	Cursor at left margin
Begfile	Cursor moved	Cursor not moved
Begline	Cursor moved	Cursor not moved
Boxstream	New mode is box mode	New mode is stream mode
Cancel	Always	Never
Cdelete	Cursor moved	Cursor not moved
Compile	Compilation successfully initiated or background compilation running	Compilation unsuccessfully initiated or background compilation not running
Copy	Always	Never
Curdate	Date inserted	Insertion would make line too long
Curday	Day inserted	Insertion would make line too long
Curfile	File inserted	Insertion would make line too long
Curfileext	File extension inserted	Insertion would make line too long
Curfilenam	File name inserted	Insertion would make line too long
Curtime	Time inserted	Insertion would make line too long
Delete	Always	Never
Down	Cursor moved	Cursor not moved
Emacscdel	Cursor moved	Cursor not moved
Emacsnewl	Always	Never
Endfile	Cursor moved	Cursor not moved
Endline	Cursor moved	Cursor not moved
Environment	Successful set or map	Syntax error or line too long
Execute	Last command successful	Last command failed
Exit	No return condition	No return condition
Graphic	Character inserted	Insertion would make line too long

Table A.4 *(continued)*

Function	Returns TRUE	Returns FALSE
Home	Cursor moved	Cursor not moved
Information	Always	Never
Initialize	Found tagged section in TOOLS.INI	Did not find tagged section in TOOLS.INI
Insert	Always	Never
Insertmode	Insert mode turned on	Insert mode turned off
Lastselect	Selection recreated	*Arg* already selected
Lasttext	Value of function eventually executed	Bad argument
Ldelete	Always	Never
Left	Cursor moved	Cursor not moved
Linsert	Always	Never
Mark	Definition/move successful	Bad argument/not found
Message	Always	Never
Meta	*Meta* turned on	*Meta* turned off
Mgrep	String found	String not found or specified, or search terminated by CTRL+BREAK, or background compilation in progress
Mlines	Movement occurred	Bad argument
Mpage	Movement occurred	Bad argument
Mpara	Cursor moved	Cursor not moved
Mreplace	Replacement successful	Replacement failed or was aborted
Msearch	String found	Bad argument/string not found
Mword	Cursor moved	Cursor not moved
Newline	Always	Never
Nextmsg	Message found	No more messages
Noedit	File or editor in no-edit state	File or editor is not in no-edit state
Paste	Almost always	Tried *Arg Arg filename Paste* and file didn't exist
Pbal	Balance successful	Bad argument/not balanced
Plines	Movement occurred	Bad argument
Ppage	Cursor moved	Cursor not moved
Ppara	Cursor moved	Cursor not moved
Print	Print successfully submitted	Could not start print job

Table A.4 *(continued)*

Function	Returns TRUE	Returns FALSE
Psearch	String found	Bad argument/string not found
Pword	Cursor moved	Cursor not moved
Qreplace	At least one replacement	String not found/invalid pattern
Quote	Almost always	Insertion would make line too long
Record	Recording turned on	Recording turned off
Refresh	File read in/deleted	Canceled, bad argument
Repeat	Function repeated and returned TRUE	Function repeated and returned FALSE, or no function to repeat
Replace	At least one replacement	String not found/invalid pattern
Restcur	Position previously saved with *Savecur*	Position not saved with *Savecur*
Right	Cursor over text of line	Cursor beyond end of line
Saveall	Always	Never
Savecur	Always	Never
Sdelete	Always	Never
Searchall	Something found	Nothing found
Setfile	File-switch successful	No alternate file, or current file needs to be saved and can't be
Setwindow	Always	Never
Shell	Shell successful	Bad argument/program not found
Sinsert	Always	Never
Tab	Cursor moved	Cursor not moved
Tell	Key pressed has function assigned	Key pressed has no function assigned
Undo	Almost always	If nothing to undo
Up	Cursor moved	Cursor not moved
Window	Successful split, join, or move	Any error

A.5 Editor Switches

Table A.5 gives an alphabetical listing of editor switches along with descriptions and default values. The first word in each description identifies the switch as a text, numeric, or Boolean switch.

Table A.5 Editor Switches

Switch	Description (and Default Value)
askexit	Boolean. Prompts for confirmation when you exit from the editor. (Default value: No)
askrtn	Boolean. Prompts you to press ENTER when returning from a *Shell* command. (Default value: Yes)
autosave	Boolean. Saves the current file whenever you switch away from it. If this switch is off, contents of file buffer are maintained, but subsequent actions, such as exiting, may lose edits. (Default value: Yes)
backup	Text. Determines what happens to the old copy of a file when a new version is saved to disk. A value of none specifies that no backup operation is to be performed; the editor simply overwrites the old file. A value of **undel** specifies that the old file is to be moved so that UNDEL.EXE can retrieve it. A value of bak specifies that the file name of the old version of the file will be changed to .BAK. (Default value: bak)
case	Boolean. Considers case to be significant for search and replace operations. For example if **case** is on, the string `Procedure` is not found as a match for the string `procedure`. (Default value: No)
displaycursor	Boolean. Shows a position on the status line in the (*row,column*) format. When off, the position listed is that of the upper-left corner. When on, the current cursor position is given. (Default value: No)
editreadonly	Boolean. Allows read-only files to be edited. If off, a read-only file is marked no-edit in the editor. (Default value: Yes)
entab	Numeric. Controls the degree to which the Microsoft Editor converts multiple spaces to tabs when reading or writing a file. A value of 0 means tabs are not used to represent white space; 1 means all multiple spaces outside of quoted strings are converted; 2 means all multiple spaces are converted to tabs. (Default value: 1)
enterinsmode	Boolean. Starts the editor up in insert mode instead of overtype mode, or switches to insert mode in the middle of an editing session. (Default value: No)
errcolor	Numeric, using hexadecimal radix. Controls the color used for error messages. The default is red text on a black background. (Default value: 04)

Table A.5 *(continued)*

Switch	Description (and Default Value)
errprompt	Boolean. Controls the "Press any key" prompt. When On, the editor stops at each error message and waits for a keystroke. (Default value: Yes)

extmake	Text. Associates a command line with a particular file extension for use by the *Compile* function. The text after the switch has this form:
	extmake:*extension commandline*
	Here, *extension* is the extension of the file to match, and *commandline* is a command line to be executed. If there is a **%s** in the command line, it is replaced with the name of the current file or with the *textarg* in the *Arg textarg Compile* command. This switch may appear more than once in the TOOLS.INI file.
	For example, you could have the following lines in TOOLS.INI:
	`extmake:bc /Z %s;`
	`extmake:for fl /c %s`
	`extmake:pas pl /c /h %s`
	`extmake:asm masm -Mx %s;`
	`extmake:c cl /c /Zep /D LINT_ARGS %s`
	`extmake:text make %s`
	The *Arg Compile* command spawns a system-level command line based on extension of current file; the editor selects the corresponding **extmake** setting. The *Arg textarg Compile* command spawns the command line for **text** extension, in which *textarg* replaces **%s**.
	This switch can also use the **%|F** syntax described in Section 7.3.

fgcolor	Numeric. Controls the color used for the editing window. The default is light gray text on a black background. (Default value: 07)
filetab	Numeric. Determines how the editor translates tabs to spaces when reading or writing a disk file. This switch also determines how the editor translates spaces to tabs for modified lines, when **entab** > 0. The value of the switch gives the number of spaces associated with each tab column. For example, the setting "filetab:4" assumes a tab column every 4 positions on each line. Every time the editor finds a tab character in a file, it loads the buffer with the number of spaces necessary to get to the next tab column. (Default value: 8)
height	Numeric. Controls the number of lines that the Microsoft Editor uses in the editing window, not including the dialog and status lines. This is useful with a nonstandard display device. Enhanced Graphics Adapter (EGA) in 43-line mode on the IBM PC uses a value of 41. Video Graphics Array (VGA) in 50-line mode uses a value of 48. (Default value: 23)
helpboldcolor	Numeric. Controls the color of text designated as boldface. (Default value: 0F)

Table A.5 *(continued)*

Switch	Description (and Default Value)
helpfiles	Text. Specifies which .HLP files should be used by on-line Help.
helpitalcolor	Numeric. Controls the color of text designated as italic. (Default value: 0A)
helpundcolor	Numeric. Controls the color of text designated as underlined. (Default value: 0C)
helpwarn-color	Numeric. Controls the color of text used for a "warning" note. Also controls the color of highlighted cross-references. (Default value: 07)
helpwindow	Boolean. Controls split-screen behavior. When off, the editor cannot split the screen to display Help information. (Default value: Yes)
hgcolor	Numeric; hexadecimal value. Controls the color of text highlighted by a search command. (See **selcolor**.) The default is green. (Default value: 02)
hike	Numeric. Specifies the cursor's new-line position (from the top of the screen) when the cursor is moved out of the current window by more than **vscroll** lines. (Default value: 4)
hscroll	Numeric. Controls the number of columns shifted left or right when the cursor is scrolled out of the editing window. (Default value: 10)
infcolor	Numeric, using hexadecimal radix. Controls the color used for informative text. The default is dark yellow text on a black background. (On some monitors, this may appear brown.) (Default value: 06)
keyboard	Text. Set to "compatible" if you have a 101-key enhanced keyboard and wish to use keyboard-enhancer programs that do not fully support the enhanced keyboard. Set to "enhanced" to restore normal operation after using "compatible." If no option is used, the editor makes its own decision about type of keyboard in use. This switch works only under DOS or real-mode OS/2.
load	Text. Specifies the name of a C-extension executable file to be loaded. Whenever this switch is assigned a new value, the extension file named is loaded into memory and initialized by calling the **WhenLoaded** function. The file named must be a full name, including base name and file extension (unless extension module is for OS/2 protected mode). See Chapter 8, "Programming C Extensions," for more information.
markfile	Text. Specifies the name of the file the Microsoft Editor searches when looking for a marker that is not in the in-memory set. This file can be created by entering lines of the following form: *markername filename line column* Here, *line* and *column* specify the position in the file *filename* where the marker *markername* appears.

Table A.5 *(continued)*

Switch	Description (and Default Value)
noise	Numeric. Controls the number of lines counted at a time when searching or loading a file. This value is displayed in the lower-right corner of the screen and may be turned off by setting **noise** to 0. (Default value: 50)
printcmd	Text. Specifies a system-level command that the editor invokes when you issue the *Print* command. For example, the following setting copies output to LPT2: `COPY %s LPT2` By default, the *Print* command sends output directly to LPT1.
readonly	Text. Specifies the DOS command invoked when the Microsoft Editor attempts to overwrite a read-only file. The current file name is appended to the command, as shown in the following example: `readonly:attrib -r %s` This command removes the read-only attribute from the current file so the file can be overwritten. If no command is specified, you are prompted to enter a new name under which to save the file.
realtabs	Boolean. Preserves actual tab characters instead of converting them to spaces. When this switch is on, the editor preserves tab alignment as characters are added and deleted, and cursor-movement functions treat each tab as a single character. (Default value: Yes)
rmargin	Numeric. Controls the right column margin used in wordwrap mode. A space typed to the right of this margin causes a line break. Wordwrap mode is turned on and off with the **wordwrap** switch. (Default value: 72)
savescreen	Boolean. Saves and restores the DOS screen (used with the *Shell* and *Exit* functions). (Default value: Yes)
searchwrap	Boolean. Causes search and replace commands to wrap past the end of the file and continue searching from the beginning. Unsuccessful searches stop after the entire file is searched once. When this switch is off, searches stop at the end of the file. (Default value: No)
selcolor	Numeric. Controls the color of text highlighted by an on-screen argument (*linearg*, *boxarg*, *streamarg*). The default is black text on a white background. Do not confuse with **hgcolor**. (Default value: 70)
shortnames	Boolean. Allows you to load a file by giving only the base name, which the editor searches for in the <information-file>. (Default value: Yes)
snow	Boolean. Eliminates snow on a CGA, at a penalty to speed. Turn this switch off if you have a CGA-compatible that doesn't generate snow. (Default value: Yes)

Table A.5 *(continued)*

Switch	Description (and Default Value)
softcr	Boolean. Attempts to indent based upon the format of the surrounding text when you invoke the *Newline* or *Emacsnewl* function. (Default value: Yes)
stacolor	Numeric. Controls the color used for the status-line information. The default is cyan text on a black background. (Default value: 03)
tabalign	Boolean. Determines where the cursor may be placed in a tab field. When off, the cursor may be placed anywhere in a tab field. When on (and if **realtabs** is also on), the cursor must align with the column position of the tab. (Default value: No)
tabdisp	Numeric. Specifies the ASCII value of the character used to display tabs. Normally, a space is used, but a graphic character can be used to show which spaces correspond to tabs. (Default value: 32)
tabstops	Numeric. Controls the number of spaces between each logical tab stop for the *Tab* and *Backtab* movement functions. Note that this switch has no relation to the interpretation of actual tabs. (Default value: 4)
tmpsav	Numeric. Controls the maximum number of recently edited files listed in the information file. If this switch is set to 0, the editor lets the information file grow without limit; all files ever edited appear in the information file until M.TMP is altered or deleted. (Default value: 20)
traildisp	Numeric. Specifies the ASCII value of the character to be displayed as trailing spaces. Note that this switch has no effect unless the **trailspace** switch is turned on. (Default value: 0)
trailspace	Boolean. Preserves trailing spaces in each line you modify. (Default value: No)
undelcount	Numeric. Controls the number of backup copies of a file that are saved when the **backup** switch is set to **undel**. When the limit is exceeded, the editor discards the oldest backup. (Default value: no limit)
undocount	Numeric. Controls the number of edit functions that you can undo. (Default value: 10)
unixre	Boolean. Specifies the use of UNIX regular-expression syntax rather than the syntax used inVersion 1.0 of the editor. (Default value: Yes)
viewonly	Boolean. Identical to the /r command-line switch and the *Noedit* function. When set, no file can be edited. (Default value: No)
vscroll	Numeric. Controls the number of lines shifted up or down when the cursor is scrolled out of the editing window. The *Mlines* and *Plines* functions also use this value. (Default value: 7)
wdcolor	Numeric. Controls the color of the border line created when you split a window. (Default value: 07).

Table A.5 *(continued)*

Switch	Description (and Default Value)
width	Numeric. Controls the width of the display mode for displays that are capable of showing more than 80 columns. Values other than 80 are supported only for a limited number of monitors. (Default value: 80)
wordwrap	Boolean. Breaks lines of text when you edit them beyond the margin specified by **rmargin**. (Default value: No)

Appendix B

Support Programs for the Microsoft Editor

This appendix discusses two programs that work in conjunction with the Microsoft Editor:

- UNDEL.EXE
- EXP.EXE

Both programs work with backup files. When a file is updated and the **backup** switch is set to **undel**, the old version of the file is copied to a hidden subdirectory called DELETED. UNDEL.EXE and EXP.EXE manipulate the files in the DELETED subdirectory.

B.1 UNDEL.EXE

Use this program to move a file from the DELETED subdirectory to the parent directory. Its command-line syntax is as follows:

undel [[*filename*]]

If *filename* is not given, the contents of the DELETED subdirectory are listed. If there is more than one version of the file, you are given a list to choose from. If the file already exists in the parent directory, the two files are swapped.

B.2 EXP.EXE

Use this program to remove all of the files in the hidden DELETED subdirectory of the specified directory. Use the following command-line syntax:

exp [[/r]] [[/q]] [[*directory*]]

If no *directory* is specified, the current directory's DELETED subdirectory is used. If the /r option is given, EXP.EXE recursively operates on all subdirectories. The /q option specifies quiet mode; the deleted file names are not displayed on the screen.

Appendix C

Microsoft Editor Messages

This appendix lists the messages that the editor can report on the dialog line, along with explanations. Some of these messages represent error conditions; in that case, the explanation given describes what went wrong and what action to take to correct the error. Other messages prompt you for information or report what operation is taking place. For example, the editor displays a message whenever it saves a file.

The messages are listed in two sections. The first section consists of messages that begin with a placeholder. A "placeholder" is a piece of information—such as a number or a file name—that varies with the situation. This section is organized alphabetically by the placeholder type. The second section consists of messages that do not begin with a placeholder. These messages are organized alphabetically by first symbol or word.

C.1 Messages Starting with Placeholders

driveletter **is an invalid drive**

You attempted to use *Setfile* to change the current drive to a drive not recognized by your system. Use a different drive letter.

file **does not exist**

You gave the *Arg Arg filename Paste* command, but the file specified does not exist. Make sure you give the complete path name of the file if it is not in the current directory.

file **does not exist. Create?**

When you started the editor, you specified a file that does not exist. Type Y to create the file or N to avoid creating the file, in which case the editor loads the most recently edited file.

file **has been changed. Refresh?**

One of the files opened for editing was altered by a program other than the editor. This change has not yet been reflected in the file buffer. You should either type Y to discard recent edits and have the editor reread the file from disk, or type N to preserve your edits. If you answer N, you should execute a save operation so that the file buffer is copied to the file stored on disk.

file **has changed. Save changes (Y/N)?**

You have attempted to exit, and the file you are currently working on has been changed since it was last saved. If you respond Y, the changes will be saved; if N, they will be discarded.

file **is read-only**

You attempted to overwrite a file that has the read-only attribute. After displaying this message, the editor prompts for a new file name under which to save the file. If you assigned a command line to the **readonly** switch, the editor asks if you want to execute this command.

'*keystroke*' **is an unknown key**

You gave an unrecognized keystroke in an assignment. Some keyboards may support keystroke combinations that the editor does not accept.

keystroke **is not assigned to any editor function**

The keystroke you pressed has no editing function assigned to it (not even the *Graphic* function, which makes keystrokes literal). You must assign a function to a keystroke before using it.

name **is not an editor function**

You attempted to assign an undefined function to a keystroke. You may have mistyped the function name, or you may have used the colon (:) instead of the definition symbol (:=) to enter a macro definition.

name **is not an editor switch**

You entered an assignment in which the left side did not contain a recognized function, macro, or switch name. You may have mistyped the name or used the wrong syntax.

number **occurrences found**

The *Searchall* function found *number* occurrences of the search string.

number **occurrences replaced**

The *Replace* or *Qreplace* function replaced *number* occurrences of the search string.

number: **Undefined opcode**

The editor detected an internal error in the way it analyzed a regular expression. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

+'*string*' not found

The *Psearch* function failed to find an occurrence of *string*. This message may be reported for both ordinary and regular-expression searches.

***value* is an illegal setting**

You attempted to give an illegal value in a switch assignment.

C.2 Other Messages

<ZFormat???>

You used the percent sign (%) in a text switch (such as **extmake**) without following it by correct syntax such as `%s` or `%F`. To include a literal use of the percent sign, enter two percent signs in a row (`%%`).

****PANIC EXIT** Really exit and lose edits?**

The editor appeared to be "hung" (non-functioning) and you pressed CTRL+BREAK five times to exit. If you answer Y, the editing session will be terminated and any unsaved changes will be lost. The editor sometimes has so many operations to perform that it appears to be hung, but is actually functioning normally. If you think this is the case, answer N to return to the editing session.

– floating point not loaded

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

– integer divide by zero

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

– not enough memory on exec

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

– not enough space for arguments

You attempted to start the editor with more file names than can be stored in free memory. This error should be impossible to get unless you use a wildcard expression (such as `*.c`) with an extraordinarily large directory.

– not enough space for environment

The editor has insufficient free memory to store the environment variables for the current session. This error should be impossible to get unless you have severely restricted the amount of memory granted to the editor.

– null pointer assignment

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

–Search for '*string*'

The *Msearch* function is currently searching for *string*.

– stack overflow

Internal function calls required a larger stack than is available to the editor. If you receive this error, it is most likely because you are using an extension that makes extensive use of local variables or function calls. Rewrite the extension so that it uses fewer or smaller local variables; then recompile and relink.

–'*string*' not found

The *Msearch* function failed to find an occurrence of *string*. This message may be reported for both ordinary and regular-expression searches.

+Search for '*string*'

The *Psearch* function is currently searching for *string*.

All edits lost (Sorry)

The editor lost information as a result of an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Are you sure you want to exit? (y/n):

You gave the *Meta Exit* command (exit without saving), or the **autosave** switch is not on and you tried to exit. Type Y to exit or N to continue the editing session.

Arg [*number*]:

You invoked the *Arg* function, which is used to introduce an argument or modify a function. The *number* placeholder indicates the number of times you invoked *Arg*. Complete the command or type *Cancel* to escape. Any *textarg*s you type in are displayed after the colon.

Arg list too long

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Argument cancelled

You invoked the *Cancel* function, which removes arguments. To clear this message from the dialog line, invoke *Cancel* again.

Argument required

The function you just invoked requires an argument.

Bad Assignment!

The editor could not perform the assignment requested. You may have mistyped your entry or requested nonexistent functions.

bad environment on exec

OS/2 only. You gave a *Compile* or *Shell* command that could not be executed.

Bad file number

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

bad format on exec

OS/2 only. You gave a *Compile* or *Shell* command that could not be executed.

Bookmark not found

You attempted to go to a file marker that does not exist or mistyped the marker name.

Can't delete *file – messagetext*

The editor could not delete the given file, for the reason given in *messagetext.*

Can't delete old version of *file*

The editor attempted to write a file to disk during the UNDEL backup procedure (the old version of the file is stored in the DELETED subdirectory), but the editor could not do the necessary file deletion. Attempt saving to an alternate file.

Can't rename *srcfile* **to** *destfile – messagetext*

The editor attempted to write a file to disk during the BAK backup procedure (the old version of the file is stored as *srcfile*.BAK), but the editor could not do the necessary renaming. Attempt saving to an alternate file.

Cannot access *file* **–** *messagetext*

You tried to load *file*, but were denied access by the system for the reason given in *messagetext.*

Cannot allocate mpPnPage – *number*

OS/2 only. The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Cannot allocate pageTmp – *number*

OS/2 only. The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Cannot change screen parameters when windows present

You attempted to reset the **height** switch when multiple windows were present. Close all windows but one before attempting to set **height**. (The editor can give this message when attempting to read TOOLS.INI.)

Cannot close this window

You attempted to close a window that cannot be merged with another window. Two windows can be merged only if the windows share one complete side. A window could not be merged with two smaller windows if each shared part of one side adjoining the larger window.

The editor closes the window if it can find at least one adjoining window that can be properly merged. If the editor cannot close the window, move to another window and close it first.

Cannot create *file* **–** *messagetext*

The editor was unable to create *file*, for the reason given in *messagetext.*

Cannot create printer intermediate file

OS/2 only. The editor was unable to create the intermediate file needed for printer queuing.

Cannot duplicate: *errormessage*

In attempting to perform a shell command or compile action, the editor was unable to redirect output. The C run-time error message indicates the nature of the problem.

Cannot find label *name*

The given *name* was the target of a jump-to-label instruction (+>, ->, or =>) within a macro, but the label was not defined. Place :>*name* in the macro to define the label.

Cannot load *file* **–** *messagetext*

You attempted to load *file* as an extensions module, but it cannot be loaded for the reason given in *messagetext.*

Cannot mgrep to <compile> during background compile

OS/2 only. You cannot start an **mgrep** during a background compile.

Cannot open *file*

The *file* was specified as an argument to the *Setfile* or *Paste* command, but the *file* cannot be opened.

Cannot open *file* **-** *errormessage*

The specified file could not be opened. The C run-time error message indicates the nature of the problem.

Cannot open *file*: *errormessage*

The specified file could not be opened. The C run-time error message indicates the nature of the problem.

Cannot open PRN: *file*

The editor could not locate or access a printer port to print *file.*

Cannot read *file*

A *file* to be read was located but could not be read.

Cannot save to directory: *directory*

You attempted to save to a directory that has the read-only attribute.

Cannot unlink file *file*: *messagetext*

The temporary file used for printing could not be deleted. The *file* placeholder is the name of the file, and *messagetext* is a C run-time error message.

Cannot write on PRN: *file*

The editor was unable to output the specified file to the printer.

Cannot write printer intermediate file

OS/2 only. The editor was unable to write to the intermediate file used for printer queuing.

Changed directory to *file*

The *Setfile* command successfully changed the current directory.

Changed drive to *file*

The *Setfile* command successfully changed the current drive.

Command could not be executed – *command*

You passed a system-level command line that could not be executed to the *Shell* or *Compile* command. This command line is displayed in the message as *command.*

Command failed to begin, *messagetext*

OS/2 only. A background compilation could not begin for the reason stated in *messagetext.*

Compilation complete – return code is *number*

DOS or real mode only. A compilation has completed with the return code shown. A nonzero return code usually indicates an error.

Compile Action *number*: **Internal Error**

The editor detected an internal error in the way it analyzed a regular expression. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Compile failed to begin, *file – messagetext*

OS/2 only. The *Compile* command could not execute the command line that it was given for the reason stated in *messagetext.*

Compile list full, try later

OS/2 only. You tried to execute too many compilations at once. The limit to the number that can be queued (that is, waiting to execute) is 16.

Cross-device link

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Delete current contents of compile log?

OS/2 only. The compile log still contains the results of a previous compilation. Type Y to delete the contents before starting another compilation.

Do you want to delete this file from the current window?

You gave the *Arg Refresh* command, which removes the file from the current window. Type Y to remove the file, or N to continue editing the file.

Do you want to replace this occurrence (Yes/No/All/Quit) :

The *Qreplace* function has found an occurrence of the search string and is prompting you for confirmation before substituting the replacement string. Type Y to make the replacement, N to avoid replacement, A to proceed with making replacements without confirmation, or Q to quit the *Qreplace* function.

Do you want to reread this file?

You invoked the *Refresh* function. Type Y to discard the most recent edits and reread the file from disk. Type any other character to cancel the function.

Do you want to save this file as *filename*?

You gave the *Arg Arg textarg Setfile* command, which saves the current file under a new file name. Type Y to proceed with the save or any other character to cancel the command.

Empty replacement string, confirm:

The *Replace* or *Qreplace* function has been invoked with an empty replacement string. Type Y to confirm or any other character to cancel. Confirming an empty replacement string means that the editor simply deletes each occurrence of the search string that it finds.

error on scratch file.

The drive on which the scratch (temporary) file resides is full. Exit the editor, change the setting of the TMP environment variable to a directory on a different drive, or free up space on the drive listed in TMP.

EstimateAction *number*: Internal Error

The editor detected an internal error in the way it analyzed a regular expression. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Exec format error

You attempted to load an extension module that does not have the correct format for this version of the editor. Rewrite the extension, relink, and recompile.

File *file* is dirty, do you want to save it?

You are trying to exit without saving a file that has been altered since it was last saved. If you answer N, the changes will be lost.

File has been deleted

The current file has been deleted by a program other than the editor, another process running under OS/2, another computer on the network, or a shelled or compiled program.

File to edit:

You started the editor without specifying a file, and there is no previously edited file. Enter a file name or press ENTER to exit back to the operating-system level.

Flushing *file* from memory

The specified *file* has been cleared from memory.

Illegal Extension Interface Called

An extension function attempted to call a low-level function that is not supported by this version of the editor. Consult Chapters 8 and 9 for a list of low-level functions supported by the editor.

Illegal setting

You attempted to set a switch to an invalid setting.

Internal Error: RE error *number*, line *number*

The editor detected an internal error in the way it analyzed a regular expression. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Invalid argument

The function just invoked does not accept the argument given. Try reentering the command using a valid argument.

Invalid drive

You attempted to execute a command with a drive letter not recognized by your system. Use a different drive letter.

Invalid pattern

You specified a regular-expression search command but gave a search string that is not a valid regular expression. Check the syntax rules for regular expressions and reenter.

Invalid replacement pattern

You specified a regular-expression search and replace command but gave an invalid replacement string. Check the syntax rules for regular-expression replacement strings and reenter.

Invoke: "*command*" (y/n)?

You attempted to overwrite a file on disk that has the read-only attribute set, and the **readonly** switch has been assigned a command. The editor asks you for confirmation before executing this command.

Kill background compile?

OS/2 only. You gave the *Arg Arg Meta Compile* command, which kills a background compilation after prompting for confirmation. Type Y to terminate the compilation or any other key to cancel the command.

Line *linenumber* too long

The indicated line exceeds the maximum of 251 characters. The editor ignores edits that exceed this limit.

line *linenumber* too long; replacement skipped

The replacement requested would have caused the indicated line to be greater than the maximum line length, so the replacement was not performed.

List Error: '*listmacro*' does not exist

The specified list macro did not exist. You may have mistyped its name or used the wrong syntax.

List Error: Nested too deeply at '*macroname*'

Macros may be nested no more than 20 deep. You tried to execute a macro that has more than 20 nesting levels; the *macroname* placeholder indicates the macro which tried to call a macro at the 21st level.

M internal error – *file*, continue?

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

macro *file* is in use

A macro attempted to redefine itself during execution (by using the *Assign* command), or a macro called a second macro that attempted to redefine the first macro.

Macros nested too deep

You created a macro that used too many levels of nesting. A macro is nested when it is executed inside another macro. Macros can be nested to 20 levels.

Mapped line *number* too long

You used the *Environment* function to replace the names of environment variables in a text string with their actual values; the resulting string was greater than 251 characters.

Math argument

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

mGetCmd called with no macros in effect

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

missing ':' in *string*

You gave an argument to the *Assign* command without including an assignment symbol (:).

Missing key assignment for '*function*'

You attempted to assign the given *function* to a keystroke but did not enter a keystroke name.

New file name:

You attempted to overwrite a file on disk that has the read-only attribute set, and the **readonly** switch has not been assigned a command. The editor asks for a new file name under which to save the file so your edits are not lost. Respond with the name of a file that can be overwritten.

Next file is *file*...

You invoked the *Setfile* function, and the editor is now loading the given file.

no alternate file

You gave the *Setfile* command file with no arguments, but there was no alternate file to load. The *Setfile* command with no arguments normally switches to the previous file.

No command to repeat

You executed the *Repeat* function to repeat the previous function, but no previous function had been executed.

No compile command known

You executed *Arg textarg Compile*, but there was no corresponding **extmake:text** definition.

No compile command known for *.ext*

No *extmake* definition has been created for the extension of the file you are currently editing.

No compile command known for extension *file*

You invoked the *Arg Compile* command, which uses the extension of the current file, but the **extmake** switch has no setting for this file extension. Before invoking *Arg Compile*, make the assignment

extmake:*ext command-line*

in which *ext* is the extension of the current file.

No compile in progress

You executed a *Compile* command without arguments, which returns the current compilation status. No compilation was in progress.

No–Edit file may not be modified

You attempted to execute an editing command that would change a No-Edit file. No-Edit files cannot be modified, even in memory.

No matching files

You invoked the *Setfile* function with a string containing the wildcards ? or *, but no file names matched this string.

No more compilation messages

The *Compile* function was given with no arguments. This function moves the cursor to the next error message. However, there are no further error messages.

No search string specified

You invoked *Psearch* or *Msearch* without a search string, and there is no previously defined search string to use as a default. Use an argument the first time you use *Psearch* or *Msearch.*

No space left on device

You attempted to save a file to a device (usually a disk drive) that has no room left. Execute a shell and make room on the drive, or else save to a different device.

No such file or directory

You attempted to use the *Setfile* function to change to a file or directory that does not exist.

No unbalanced characters found

You invoked the *Pbal* function, but there are no unbalanced braces, brackets, or parentheses.

No-Execute Record Mode - Press *keystroke* to resume normal editing

You have completed the special macro recording function (*Meta Record*) that records the macro without executing the keystrokes. This message reminds you that the editor is not hung. You may continue normal editing after pressing the specified key.

Not enough core

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Not enough free memory

DOS or real mode only. The editor was unable to allocate enough memory for all the files that were opened and was unable to recover. Normally, the editor recovers from lack of memory by closing files if it can. Terminate the editing session immediately.

Not enough memory for pattern

You specified a regular-expression pattern too complicated for the editor to implement. Specify a simpler pattern.

not enough memory on exec

OS/2 only. You gave a *Compile* or *Shell* command that could not be executed.

Not enough room for macro *macroname*

You defined 1,024 macros and cannot add another. To reduce the number of macros, exit the editor and restart.

Not supported by video display

You attempted to assign the **height** switch to a value not supported by the video adapter card. Only the following values are supported: 23 for CGA and monochrome; 23 and 41 for EGA; and 23, 41, and 48 for VGA.

Nothing to ReDo

You have repeated the last command in the editing command list; there are no more commands that can be repeated.

Nothing to UnDo

You have reversed every editing command that remains in the UnDo list; there are no more editing commands to reverse.

Out of disk space for VM swap file – recovering

DOS or real mode only. An insufficient amount of space was available for swapping. Exit the editor and reset the TMP environment variable to a disk drive with more space, if possible.

Out of far memory – recovering

The editor has insufficient memory to support all the files open and is now trying to recover memory so that no information is lost.

Out of local heap – recovering

The editor has insufficient memory to support all the files open and is now trying to recover memory so that no information is lost.

Out of local memory

DOS or real mode only. The editor was unable to allocate enough memory for all the files that were opened and was unable to recover. Normally, the editor recovers from lack of memory by closing files if it can. Terminate the editing session and do not open as many files at the same time.

Out of space on *device*

You attempted to save a file to a *device* (usually a disk drive) with no room left. Create free space on the drive, or else save to a different device.

Packed file is corrupt

DOS or real mode only. You used the *Shell* or *Compile* function to execute a corrupted packed file.

Permission denied

You attempted to write to a locked file.

Please strike any key to continue

The *Shell* command has terminated, and the editor is ready to continue the normal editing session.

Press any key...

The editor has completed some operation and is waiting for you to press a key before resuming the editing session.

Print list full, try later

OS/2 only. There is no space in the printing queue for another file. Wait, then submit your print request again.

Printing *file*... Press Esc to abort

Your print request has been accepted. Press ESC to cancel the printing.

Printing: *file*

The specified *file* will be printed.

Query search string:

You invoked the *Qreplace* function. Enter a new search string, press ENTER (or invoke the *Emacsnewl* function) to accept the old search string, or invoke *Cancel* to exit.

Queued: '*file*'

OS/2 only. You gave a new compile command before the last compilation was complete. The editor will not spawn more than one compilation at a time but can hold up to 16 compile commands to be executed sequentially.

RE emulator stack overflow

The editor encountered an internal error while evaluating a regular expression. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

RemoveFile can't find file

An extension module passed a nonexistent file to the **RemoveFile** low-level function.

Removing file *filename*

The **RemoveFile** low-level function has removed a file buffer from memory. This is not an error unless you need to have the file open for editing.

Replace string:

You invoked the *Replace* function. Enter a new replacement string, press ENTER (or invoke the *Emacsnewl* function) to accept the old replacement string, or invoke *Cancel* to exit.

Replace this occurrence (Yes/No/All/Quit):

You executed the *Qreplace* function, which prompts for the replacement rather than performing the replacement automatically. You can choose not to replace the text, to replace the text for this occurrence only, to replace all following occurrences, or to quit the *Qreplace* function.

Resource deadlock would occur

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Result too large

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

run-time error

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Save all remaining changed files (Y/N)?

This prompt appears when there are one or more changed files in addition to the file you were working on when you attempted to exit. If you respond Y, all the remaining changed files will be saved. If you respond N, you will be prompted about saving each file individually.

Save file *file* before flushing (y/n/p)

The editor is running low on memory and needs to remove file buffers. No saved files are available for removal, and the editor must remove unsaved files. Type Y to save changes to *file* before it is removed, or N to remove *file* without saving changes. Type P to save all files that have unsaved changes. Invoke the *Cancel* function to prevent *file* from being removed. This last option causes the editor to try to remove a different *file* if one is available.

Saving *file* ...

You invoked the *Setfile* function, and the editor is now saving *file*.

Search string:

You invoked the *Replace* function. Enter a new search string, press ENTER (or invoke the *Emacsnewl* function) to accept the old search string, or invoke *Cancel* to exit.

Source file not found: *file*

The *Arg Compile* command was invoked but could not find one of the files specified. Reset the **extmake** switch so that compilations can run successfully.

Spawn failed on *command – messagetext*

The *Shell* or *Compile* function was given a command that could not be executed for the reason given in *messagetext.*

Swapping file *file* **already exists.**

DOS or real mode only. When you started the editor, the virtual-memory swapping file already existed. (On start-up, the editor assumes that this file has been deleted.) Delete *file* and restart the editor.

Testing VM error at *address* **error on scratch file. This usually indicates a full-disk on the scratch file device. Please attempt to free some space on it.**

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

This program cannot be run in DOS mode.

OS/2 only. You invoked the *Shell* or *Compile* command, which can be run in real mode only (DOS or 3. *x* compatibility box).

Too many open files

The editor tried to open more files at the operating-system level, but the system cannot open any more files. Typically, this error occurs when an extension opens many files, or the FILES setting in CONFIG.SYS is too small. Increase the FILES setting in CONFIG.SYS if this setting is not already at the system maximum, then reboot. (This maximum is 20 in most versions of DOS.)

Too many windows

You tried to open more than eight windows, which is the maximum number supported by the editor.

Unable to create pipe:

OS/2 only. This message appears only on start-up. The editor cannot generate the pipes it needs for background compiling and printing. These features will not be available during the current editing session. This problem is probably the result of a lack of memory or file handles.

Unable to open swapping file *file – messagetext*

DOS or real mode only. When you started the editor, it was unable to recreate the virtual-memory swapping file. Make sure your TMP environment variable lists an accessible location.

Unable to read TOOLS.INI

The editor could not read the TOOLS.INI file upon start-up. Make sure that the TOOLS.INI file read permission is in the proper format.

Unable to recover – abort now?

The editor has run out of memory and exhausted all possible roads to recovery. (The editor takes a number of measures to recover memory before it reports this error.) If you have files you need to save, type N to save them now before the editor terminates. Otherwise, type Y to terminate immediately.

Unable to set up compile pipe

OS/2 only. The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Unable to start compile thread

OS/2 only. The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Unable to start Idle thread

OS/2 only. The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Unable to start printing thread

OS/2 only. The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Unknown error

The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

unknown function *name*

You attempted to execute a macro-definition list containing an undefined function or macro name. Make sure that *name* is defined before appearing in a macro or as an argument to *Execute*.

Unsupported video mode. Please change modes and restart.

You started the editor in an unrecognized video mode. Please exit the editor and reset the operating system video mode.

VM: Alloc Free

OS/2 only. The editor encountered an internal error. Please note the circumstances of this error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Warning: continuation character on last line!

The last line from the appropriate section of TOOLS.INI ended in a continuation character but there was no following line.

Window too small to split

You attempted to create a window smaller than 5 lines or 5 columns. Choose a different cursor position to split the window.

You have more files to edit. Are you sure you want to exit? (y/n):

You specified more than one file when you started the editor, but you are quitting without having edited all of them. Type Y to exit or N to continue editing.

You have unsaved files. Are you sure you want to exit? (y/n):

You gave the *Meta Exit* command, and the editor has detected at least one file open with unsaved changes. Type Y to exit without saving, or any other key to continue editing.

Glossary

This glossary defines terms in this manual used in a technical or unique way.

Arg A function modifier that introduces an argument or an editing function. The argument may be of any type and is passed to the next function as input. For example, the command *Arg textarg Copy* passes the argument *textarg* to the function *Copy*.

argument An input to a function. The Microsoft Editor uses two classes of arguments: cursor-movement arguments and text arguments. Cursor-movement arguments (*boxarg*, *linearg*, and *streamarg*) specify a range of characters on the screen. Text arguments (*markarg*, *numarg*, and *textarg*) are typed in directly. Arguments are introduced by using the *Arg* function.

assignment *See* "function assignment."

box mode The editing mode that lets you highlight cursor-movement arguments as rectangular areas of text.

boxarg A rectangular area on the screen, defined by the two opposite corners: the initial cursor position and the current cursor position. The editor must be in box mode, and the two cursor positions must be on separate rows and separate columns. A *boxarg* is generated by invoking the *Arg* function and then moving the cursor to a new location.

buffer An area in memory in which a copy of the file is kept and changed as you edit. This buffer is copied to disk when you do a save operation.

C extension A C-language module that defines new editing functions and switches. *See* Chapter 8.

Clipboard A pseudo file that holds text selected with the *Copy*, *Ldelete*, or *Sdelete* functions. You can use the *Paste* function to insert text from the Clipboard into a file.

configuration A description of the specific assignments of functions to keys. For example, a BRIEF configuration implies that the Microsoft Editor uses keys similar to those that the BRIEF editor uses to invoke similar functions.

cursor The blinking white underline character that defines current position. When you type characters into the file, they appear at the cursor position.

default A setting assumed by the editor until you specify otherwise. The Microsoft Editor uses two categories of default settings: function assignments and switches.

dialog line The line just above the status line. The dialog line reports editor messages. You enter text arguments on this line.

emacs A popular mainframe editor from which the functions *Emacscdel* and *Emacsnewl* were taken.

file history A list of the most recently edited files, including any files currently open for editing. The file history is displayed in the <information-file> pseudo file.

function assignment A method of assigning an editor function to a specific keystroke so that pressing the keystroke invokes the function. Use the *Arg textarg Assign* command to make an assignment for a single editing session, or you can enter the assignment in the TOOLS.INI file so that it may be used during any editing session. *See* Chapter 6.

graphic *See* "literal."

initial cursor position The position the cursor is in when the *Arg* function is invoked.

insert mode An input mode that inserts rather than replaces characters in the file as they are entered.

linearg A range of complete lines, including all the lines from the initial cursor position to the current cursor position. The editor must be in box mode. You define a *linearg* by invoking the *Arg* function (pressing ALT+A), then moving the cursor to a different line but same column as the initial cursor position.

literal The corresponding ASCII value of a keystroke. When a keystroke is considered literal or graphic, pressing the key causes the editor to place the corresponding value in a file.

macro A function made up of arguments and previously defined functions. For example, you can create a macro that contains a set of repeatedly performed functions and assign the macro to a keystroke. Those functions can be carried out much more quickly and simply by invoking the macro. *See* Chapter 6.

markarg A special type of *textarg* that has been previously defined to be a marker—that is, it is associated with a particular position in the file.

marker A name assigned to a cursor position in a file so that this position can be referred to within a command by using this name. For example, the *Arg markarg Copy* command selects the text between the cursor position and the marker position. A marker is assigned using the *Arg Arg textarg Mark* command.

Meta A function that modifies other functions so they perform differently. The effect of *Meta* varies with each function.

nested macro A macro executed by another macro.

numarg A numerical value you enter on the dialog line, which is passed to a function. A *numarg* is introduced by the *Arg* function.

overtype mode The input mode that replaces characters at the cursor position instead of inserting them. Unlike insert mode, this mode does not preserve existing text at the cursor position.

pseudo file A file kept in memory that does not correspond to a disk file. The name of a pseudo file is always enclosed in angle brackets (<>). You can create your pseudo files for fast saving of temporary information. In addition, the editor creates several pseudo files of its own and gives them special meaning. *See* Chapter 4.

recorded macro A macro you create with the *Record* function. The *Record* function turns record mode on and off. All actions you execute while in record mode are by default assigned to the RECORDVALUE macro. *See* Chapter 6.

regular expression A pattern specifying a set of one or more character strings to search for. It may be a simple string or a more complex arrangement of characters and special symbols that specify a variety of strings to be matched. *See* Chapter 5.

return value A value returned by an editing function. The value may be true or false, depending on whether the function was successful. This value can be used to create complex macros that perform differently depending upon the results of individual functions within the macro. *See* Chapter 6.

short name A file name without a path or file extension. The *Setfile* and *Paste* functions can take a short name as input and search for a full-length file name in the file history that contains the short name.

status line The bottom line of the screen. This line contains useful information about the current editing session.

stream mode The editing mode that lets you highlight cursor-movement arguments as continuous streams of text, rather than rectangular areas.

streamarg The contiguous stream of characters between two positions—the initial cursor position and the new cursor position. The editor must be in stream mode. You define a *streamarg* by invoking *Arg*, moving the cursor to any new location, and then invoking *Sinsert* or *Sdelete*.

switch A variable that modifies the way the editor performs. The Microsoft Editor uses three kinds of switches: Boolean switches, which turn a certain editor feature on or off; numeric switches, which specify numeric constants; and text switches, which specify a string of characters. *See* Chapter 7.

tag An identifier enclosed in brackets ([]) that marks the beginning of a section in the TOOLS.INI file. For example, unless you rename the editor, the principal section for editor settings starts at the [M] tag. *See* Chapter 7.

textarg A string of text that you enter on the dialog line after invoking the *Arg* function. The text that you enter is passed as input to the next function.

TOOLS.INI A file that contains initialization information for the Microsoft Editor and other programs. The file is divided into sections with the use of tags. These sections can be loaded automatically when the editor is started or by command from within the editor. *See* Chapter 7.

window An area on the screen used to display part of a file. Unless a file is extremely small, it is impossible to see all of it on the screen at once. Therefore you see a portion of the file through the main editing window at any one time. It is possible to see any part of the file by moving or scrolling this window. The Microsoft Editor allows you to open multiple windows on the screen, using the *Window* function, for viewing different parts of the same file or different files.

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft

0689 Part No. 07823